



Standard ECMA-262

12th Edition / June 2021

**ECMAScript[®] 2021
Language Specification**

Standard

Ecma International
Rue du Rhone 114
CH-1204 Geneva
Tel: +41 22 849 6000
Fax: +41 22 849 6001
Web: <https://www.ecma-international.org>



COPYRIGHT PROTECTED DOCUMENT

COPYRIGHT NOTICE

© 2020 Ecma International

This document may be copied, published and distributed to others, and certain derivative works of it may be prepared, copied, published, and distributed, in whole or in part, provided that the above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:

- (i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),*
- (ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,*
- (iii) translations of this document into languages other than English and into different formats and*
- (iv) works by making use of this specification in standard conformant products by implementing (e.g. by copy and paste wholly or partly) the functionality therein.*

However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.

The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.

The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns.

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."

Software License

All Software contained in this document ("Software") is protected by copyright and is being made available under the "BSD License", included below. This Software may be subject to third party rights (rights from parties other than Ecma International), including patent rights, and no licenses under such third party rights are granted under this license even if the third party concerned is a member of Ecma International. SEE THE ECMA CODE OF CONDUCT IN PATENT MATTERS AVAILABLE AT <https://www.ecma-international.org/memento/codeofconduct.htm> FOR INFORMATION REGARDING THE LICENSING OF PATENT CLAIMS THAT ARE REQUIRED TO IMPLEMENT ECMA INTERNATIONAL STANDARDS*.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the authors nor Ecma International may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE ECMA INTERNATIONAL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ECMA INTERNATIONAL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ECMAScript® 2021 Language Specification



About this Specification

The document at <https://tc39.es/ecma262/> is the most accurate and up-to-date ECMAScript specification. It contains the content of the most recent yearly snapshot plus any [finished proposals](#) (those that have reached Stage 4 in the [proposal process](#) and thus are implemented in several implementations and will be in the next practical revision) since that snapshot was taken.

Contributing to this Specification

This specification is developed on GitHub with the help of the ECMAScript community. There are a number of ways to contribute to the development of this specification:

GitHub Repository: <https://github.com/tc39/ecma262>

Issues: [All Issues](#), [File a New Issue](#)

Pull Requests: [All Pull Requests](#), [Create a New Pull Request](#)

Test Suite: [Test262](#)

Editors:

- [Jordan Harband \(@ljharb\)](#)
- [Shu-yu Guo \(@_shu\)](#)
- [Michael Ficarra \(@smooshMap\)](#)
- [Kevin Gibbons \(@bakkoting\)](#)

Community:

- Discourse: <https://es.discourse.group>
- IRC: [#tc39](#) on [freenode](#)
- Mailing List Archives: <https://esdiscuss.org/>

Refer to the [colophon](#) for more information on how this document is created.

Table of Contents

Introduction

1 Scope

2 Conformance

2.1 Example Clause Heading

3 Normative References

4 Overview

4.1 Web Scripting

4.2 Hosts and Implementations

4.3 ECMAScript Overview

4.3.1 Objects

4.3.2 The Strict Variant of ECMAScript

4.4 Terms and Definitions

4.4.1 implementation-approximated

4.4.2 implementation-defined

4.4.3 host-defined

4.4.4 type

4.4.5 primitive value

4.4.6 object

4.4.7 constructor

4.4.8 prototype

4.4.9 ordinary object

4.4.10 exotic object

4.4.11 standard object

4.4.12 built-in object

4.4.13 undefined value

4.4.14 Undefined type

4.4.15 null value

4.4.16 Null type

4.4.17 Boolean value

4.4.18 Boolean type

4.4.19 Boolean object

4.4.20 String value

4.4.21 String type

4.4.22 String object

4.4.23 Number value

4.4.24 Number type

4.4.25 Number object

4.4.26 Infinity

4.4.27 NaN

4.4.28 BigInt value

4.4.29 BigInt type

4.4.30 BigInt object

4.4.31 Symbol value

- 4.4.32 Symbol type
- 4.4.33 Symbol object
- 4.4.34 function
- 4.4.35 built-in function
- 4.4.36 property
- 4.4.37 method
- 4.4.38 built-in method
- 4.4.39 attribute
- 4.4.40 own property
- 4.4.41 inherited property
- 4.5 Organization of This Specification
- 5 Notational Conventions
 - 5.1 Syntactic and Lexical Grammars
 - 5.1.1 Context-Free Grammars
 - 5.1.2 The Lexical and RegExp Grammars
 - 5.1.3 The Numeric String Grammar
 - 5.1.4 The Syntactic Grammar
 - 5.1.5 Grammar Notation
 - 5.2 Algorithm Conventions
 - 5.2.1 Abstract Operations
 - 5.2.2 Syntax-Directed Operations
 - 5.2.3 Runtime Semantics
 - 5.2.3.1 Implicit Completion Values
 - 5.2.3.2 Throw an Exception
 - 5.2.3.3 ReturnIfAbrupt
 - 5.2.3.4 ReturnIfAbrupt Shorthands
 - 5.2.4 Static Semantics
 - 5.2.5 Mathematical Operations
 - 5.2.6 Value Notation
- 6 ECMAScript Data Types and Values
 - 6.1 ECMAScript Language Types
 - 6.1.1 The Undefined Type
 - 6.1.2 The Null Type
 - 6.1.3 The Boolean Type
 - 6.1.4 The String Type
 - 6.1.4.1 `StringIndexOf (string, searchValue, fromIndex)`
 - 6.1.5 The Symbol Type
 - 6.1.5.1 Well-Known Symbols
 - 6.1.6 Numeric Types
 - 6.1.6.1 The Number Type
 - 6.1.6.1.1 `Number::unaryMinus (x)`
 - 6.1.6.1.2 `Number::bitwiseNOT (x)`
 - 6.1.6.1.3 `Number::exponentiate (base, exponent)`
 - 6.1.6.1.4 `Number::multiply (x, y)`
 - 6.1.6.1.5 `Number::divide (x, y)`
 - 6.1.6.1.6 `Number::remainder (n, d)`
 - 6.1.6.1.7 `Number::add (x, y)`
 - 6.1.6.1.8 `Number::subtract (x, y)`

- 6.1.6.1.9 Number::leftShift (*x*, *y*)
- 6.1.6.1.10 Number::signedRightShift (*x*, *y*)
- 6.1.6.1.11 Number::unsignedRightShift (*x*, *y*)
- 6.1.6.1.12 Number::lessThan (*x*, *y*)
- 6.1.6.1.13 Number::equal (*x*, *y*)
- 6.1.6.1.14 Number::sameValue (*x*, *y*)
- 6.1.6.1.15 Number::sameValueZero (*x*, *y*)
- 6.1.6.1.16 NumberBitwiseOp (*op*, *x*, *y*)
- 6.1.6.1.17 Number::bitwiseAND (*x*, *y*)
- 6.1.6.1.18 Number::bitwiseXOR (*x*, *y*)
- 6.1.6.1.19 Number::bitwiseOR (*x*, *y*)
- 6.1.6.1.20 Number::toString (*x*)
- 6.1.6.2 The BigInt Type
 - 6.1.6.2.1 BigInt::unaryMinus (*x*)
 - 6.1.6.2.2 BigInt::bitwiseNOT (*x*)
 - 6.1.6.2.3 BigInt::exponentiate (*base*, *exponent*)
 - 6.1.6.2.4 BigInt::multiply (*x*, *y*)
 - 6.1.6.2.5 BigInt::divide (*x*, *y*)
 - 6.1.6.2.6 BigInt::remainder (*n*, *d*)
 - 6.1.6.2.7 BigInt::add (*x*, *y*)
 - 6.1.6.2.8 BigInt::subtract (*x*, *y*)
 - 6.1.6.2.9 BigInt::leftShift (*x*, *y*)
 - 6.1.6.2.10 BigInt::signedRightShift (*x*, *y*)
 - 6.1.6.2.11 BigInt::unsignedRightShift (*x*, *y*)
 - 6.1.6.2.12 BigInt::lessThan (*x*, *y*)
 - 6.1.6.2.13 BigInt::equal (*x*, *y*)
 - 6.1.6.2.14 BigInt::sameValue (*x*, *y*)
 - 6.1.6.2.15 BigInt::sameValueZero (*x*, *y*)
 - 6.1.6.2.16 BinaryAnd (*x*, *y*)
 - 6.1.6.2.17 BinaryOr (*x*, *y*)
 - 6.1.6.2.18 BinaryXor (*x*, *y*)
 - 6.1.6.2.19 BigIntBitwiseOp (*op*, *x*, *y*)
 - 6.1.6.2.20 BigInt::bitwiseAND (*x*, *y*)
 - 6.1.6.2.21 BigInt::bitwiseXOR (*x*, *y*)
 - 6.1.6.2.22 BigInt::bitwiseOR (*x*, *y*)
 - 6.1.6.2.23 BigInt::toString (*x*)
- 6.1.7 The Object Type
 - 6.1.7.1 Property Attributes
 - 6.1.7.2 Object Internal Methods and Internal Slots
 - 6.1.7.3 Invariants of the Essential Internal Methods
 - 6.1.7.4 Well-Known Intrinsic Objects
- 6.2 ECMAScript Specification Types
 - 6.2.1 The List and Record Specification Types
 - 6.2.2 The Set and Relation Specification Types
 - 6.2.3 The Completion Record Specification Type
 - 6.2.3.1 Await
 - 6.2.3.1.1 Await Fulfilled Functions
 - 6.2.3.1.2 Await Rejected Functions

- 6.2.3.2 NormalCompletion
- 6.2.3.3 ThrowCompletion
- 6.2.3.4 UpdateEmpty (*completionRecord, value*)
- 6.2.4 The Reference Record Specification Type
 - 6.2.4.1 IsPropertyReference (*V*)
 - 6.2.4.2 IsUnresolvableReference (*V*)
 - 6.2.4.3 IsSuperReference (*V*)
 - 6.2.4.4 GetValue (*V*)
 - 6.2.4.5 PutValue (*V, W*)
 - 6.2.4.6 GetThisValue (*V*)
 - 6.2.4.7 InitializeReferencedBinding (*V, W*)
- 6.2.5 The Property Descriptor Specification Type
 - 6.2.5.1 IsAccessorDescriptor (*Desc*)
 - 6.2.5.2 IsDataDescriptor (*Desc*)
 - 6.2.5.3 IsGenericDescriptor (*Desc*)
 - 6.2.5.4 FromPropertyDescriptor (*Desc*)
 - 6.2.5.5 ToPropertyDescriptor (*Obj*)
 - 6.2.5.6 CompletePropertyDescriptor (*Desc*)
- 6.2.6 The Environment Record Specification Type
- 6.2.7 The Abstract Closure Specification Type
- 6.2.8 Data Blocks
 - 6.2.8.1 CreateByteDataBlock (*size*)
 - 6.2.8.2 CreateSharedByteDataBlock (*size*)
 - 6.2.8.3 CopyDataBlockBytes (*toBlock, toIndex, fromBlock, fromIndex, count*)
- 7 Abstract Operations
 - 7.1 Type Conversion
 - 7.1.1 ToPrimitive (*input* [, *preferredType*])
 - 7.1.1.1 OrdinaryToPrimitive (*O, hint*)
 - 7.1.2 ToBoolean (*argument*)
 - 7.1.3 ToNumeric (*value*)
 - 7.1.4 ToNumber (*argument*)
 - 7.1.4.1 ToNumber Applied to the String Type
 - 7.1.4.1.1 RS: MV
 - 7.1.5 ToIntegerOrInfinity (*argument*)
 - 7.1.6 ToInt32 (*argument*)
 - 7.1.7 ToUint32 (*argument*)
 - 7.1.8 ToInt16 (*argument*)
 - 7.1.9 ToUint16 (*argument*)
 - 7.1.10 ToInt8 (*argument*)
 - 7.1.11 ToUint8 (*argument*)
 - 7.1.12 ToUint8Clamp (*argument*)
 - 7.1.13 ToBigInt (*argument*)
 - 7.1.14 StringToBigInt (*argument*)
 - 7.1.15 ToBigInt64 (*argument*)
 - 7.1.16 ToBigUint64 (*argument*)
 - 7.1.17 ToString (*argument*)
 - 7.1.18 ToObject (*argument*)
 - 7.1.19 ToPropertyKey (*argument*)

- 7.1.20 ToLength (*argument*)
- 7.1.21 CanonicalNumericIndexString (*argument*)
- 7.1.22 ToIndex (*value*)
- 7.2 Testing and Comparison Operations
 - 7.2.1 RequireObjectCoercible (*argument*)
 - 7.2.2 IsArray (*argument*)
 - 7.2.3 IsCallable (*argument*)
 - 7.2.4 IsConstructor (*argument*)
 - 7.2.5 IsExtensible (*O*)
 - 7.2.6 IsIntegralNumber (*argument*)
 - 7.2.7 IsPropertyKey (*argument*)
 - 7.2.8 IsRegExp (*argument*)
 - 7.2.9 IsStringPrefix (*p, q*)
 - 7.2.10 SameValue (*x, y*)
 - 7.2.11 SameValueZero (*x, y*)
 - 7.2.12 SameValueNonNumeric (*x, y*)
 - 7.2.13 Abstract Relational Comparison
 - 7.2.14 Abstract Equality Comparison
 - 7.2.15 Strict Equality Comparison
- 7.3 Operations on Objects
 - 7.3.1 MakeBasicObject (*internalSlotsList*)
 - 7.3.2 Get (*O, P*)
 - 7.3.3 GetV (*V, P*)
 - 7.3.4 Set (*O, P, V, Throw*)
 - 7.3.5 CreateDataProperty (*O, P, V*)
 - 7.3.6 CreateMethodProperty (*O, P, V*)
 - 7.3.7 CreateDataPropertyOrThrow (*O, P, V*)
 - 7.3.8 DefinePropertyOrThrow (*O, P, desc*)
 - 7.3.9 DeletePropertyOrThrow (*O, P*)
 - 7.3.10 GetMethod (*V, P*)
 - 7.3.11 HasProperty (*O, P*)
 - 7.3.12 HasOwnProperty (*O, P*)
 - 7.3.13 Call (*F, V [, argumentsList]*)
 - 7.3.14 Construct (*F [, argumentsList [, newTarget]]*)
 - 7.3.15 SetIntegrityLevel (*O, level*)
 - 7.3.16 TestIntegrityLevel (*O, level*)
 - 7.3.17 CreateArrayFromList (*elements*)
 - 7.3.18 LengthOfArrayLike (*obj*)
 - 7.3.19 CreateListFromArrayLike (*obj [, elementTypes]*)
 - 7.3.20 Invoke (*V, P [, argumentsList]*)
 - 7.3.21 OrdinaryHasInstance (*C, O*)
 - 7.3.22 SpeciesConstructor (*O, defaultConstructor*)
 - 7.3.23 EnumerableOwnPropertyNames (*O, kind*)
 - 7.3.24 GetFunctionRealm (*obj*)
 - 7.3.25 CopyDataProperties (*target, source, excludedItems*)
- 7.4 Operations on Iterator Objects
 - 7.4.1 GetIterator (*obj [, hint [, method]]*)
 - 7.4.2 IteratorNext (*iteratorRecord [, value]*)

- 7.4.3 IteratorComplete (*iterResult*)
- 7.4.4 IteratorValue (*iterResult*)
- 7.4.5 IteratorStep (*iteratorRecord*)
- 7.4.6 IteratorClose (*iteratorRecord, completion*)
- 7.4.7 AsyncIteratorClose (*iteratorRecord, completion*)
- 7.4.8 CreateIterResultObject (*value, done*)
- 7.4.9 CreateListIteratorRecord (*list*)
- 7.4.10 IterableToList (*items [, method]*)

8 Syntax-Directed Operations

8.1 Scope Analysis

- 8.1.1 SS: BoundNames
- 8.1.2 SS: DeclarationPart
- 8.1.3 SS: IsConstantDeclaration
- 8.1.4 SS: LexicallyDeclaredNames
- 8.1.5 SS: LexicallyScopedDeclarations
- 8.1.6 SS: VarDeclaredNames
- 8.1.7 SS: VarScopedDeclarations
- 8.1.8 SS: TopLevelLexicallyDeclaredNames
- 8.1.9 SS: TopLevelLexicallyScopedDeclarations
- 8.1.10 SS: TopLevelVarDeclaredNames
- 8.1.11 SS: TopLevelVarScopedDeclarations

8.2 Labels

- 8.2.1 SS: ContainsDuplicateLabels
- 8.2.2 SS: ContainsUndefinedBreakTarget
- 8.2.3 SS: ContainsUndefinedContinueTarget

8.3 Function Name Inference

- 8.3.1 SS: HasName
- 8.3.2 SS: IsFunctionDefinition
- 8.3.3 SS: IsAnonymousFunctionDefinition (*expr*)
- 8.3.4 SS: IsIdentifierRef
- 8.3.5 RS: NamedEvaluation

8.4 Contains

- 8.4.1 SS: Contains
- 8.4.2 SS: ComputedPropertyContains

8.5 Miscellaneous

- 8.5.1 RS: InstantiateFunctionObject
- 8.5.2 RS: BindingInitialization
 - 8.5.2.1 InitializeBoundName (*name, value, environment*)
- 8.5.3 RS: IteratorBindingInitialization
- 8.5.4 SS: AssignmentTargetType
- 8.5.5 SS: PropName

9 Executable Code and Execution Contexts

9.1 Environment Records

- 9.1.1 The Environment Record Type Hierarchy
 - 9.1.1.1 Declarative Environment Records
 - 9.1.1.1.1 HasBinding (*N*)
 - 9.1.1.1.2 CreateMutableBinding (*N, D*)
 - 9.1.1.1.3 CreateImmutableBinding (*N, S*)

- 9.1.1.1.4 InitializeBinding (*N*, *V*)
- 9.1.1.1.5 SetMutableBinding (*N*, *V*, *S*)
- 9.1.1.1.6 GetBindingValue (*N*, *S*)
- 9.1.1.1.7 DeleteBinding (*N*)
- 9.1.1.1.8 HasThisBinding ()
- 9.1.1.1.9 HasSuperBinding ()
- 9.1.1.1.10 WithBaseObject ()
- 9.1.1.2 Object Environment Records
 - 9.1.1.2.1 HasBinding (*N*)
 - 9.1.1.2.2 CreateMutableBinding (*N*, *D*)
 - 9.1.1.2.3 CreateImmutableBinding (*N*, *S*)
 - 9.1.1.2.4 InitializeBinding (*N*, *V*)
 - 9.1.1.2.5 SetMutableBinding (*N*, *V*, *S*)
 - 9.1.1.2.6 GetBindingValue (*N*, *S*)
 - 9.1.1.2.7 DeleteBinding (*N*)
 - 9.1.1.2.8 HasThisBinding ()
 - 9.1.1.2.9 HasSuperBinding ()
 - 9.1.1.2.10 WithBaseObject ()
- 9.1.1.3 Function Environment Records
 - 9.1.1.3.1 BindThisValue (*V*)
 - 9.1.1.3.2 HasThisBinding ()
 - 9.1.1.3.3 HasSuperBinding ()
 - 9.1.1.3.4 GetThisBinding ()
 - 9.1.1.3.5 GetSuperBase ()
- 9.1.1.4 Global Environment Records
 - 9.1.1.4.1 HasBinding (*N*)
 - 9.1.1.4.2 CreateMutableBinding (*N*, *D*)
 - 9.1.1.4.3 CreateImmutableBinding (*N*, *S*)
 - 9.1.1.4.4 InitializeBinding (*N*, *V*)
 - 9.1.1.4.5 SetMutableBinding (*N*, *V*, *S*)
 - 9.1.1.4.6 GetBindingValue (*N*, *S*)
 - 9.1.1.4.7 DeleteBinding (*N*)
 - 9.1.1.4.8 HasThisBinding ()
 - 9.1.1.4.9 HasSuperBinding ()
 - 9.1.1.4.10 WithBaseObject ()
 - 9.1.1.4.11 GetThisBinding ()
 - 9.1.1.4.12 HasVarDeclaration (*N*)
 - 9.1.1.4.13 HasLexicalDeclaration (*N*)
 - 9.1.1.4.14 HasRestrictedGlobalProperty (*N*)
 - 9.1.1.4.15 CanDeclareGlobalVar (*N*)
 - 9.1.1.4.16 CanDeclareGlobalFunction (*N*)
 - 9.1.1.4.17 CreateGlobalVarBinding (*N*, *D*)
 - 9.1.1.4.18 CreateGlobalFunctionBinding (*N*, *V*, *D*)
- 9.1.1.5 Module Environment Records
 - 9.1.1.5.1 GetBindingValue (*N*, *S*)
 - 9.1.1.5.2 DeleteBinding (*N*)
 - 9.1.1.5.3 HasThisBinding ()
 - 9.1.1.5.4 GetThisBinding ()

- 9.1.1.5.5 CreateImportBinding (*N, M, N2*)
 - 9.1.2 Environment Record Operations
 - 9.1.2.1 GetIdentifierReference (*env, name, strict*)
 - 9.1.2.2 NewDeclarativeEnvironment (*E*)
 - 9.1.2.3 NewObjectEnvironment (*O, E*)
 - 9.1.2.4 NewFunctionEnvironment (*F, newTarget*)
 - 9.1.2.5 NewGlobalEnvironment (*G, thisValue*)
 - 9.1.2.6 NewModuleEnvironment (*E*)
 - 9.2 Realms
 - 9.2.1 CreateRealm ()
 - 9.2.2 CreateIntrinsics (*realmRec*)
 - 9.2.3 SetRealmGlobalObject (*realmRec, globalObj, thisValue*)
 - 9.2.4 SetDefaultGlobalBindings (*realmRec*)
 - 9.3 Execution Contexts
 - 9.3.1 GetActiveScriptOrModule ()
 - 9.3.2 ResolveBinding (*name [, env]*)
 - 9.3.3 GetThisEnvironment ()
 - 9.3.4 ResolveThisBinding ()
 - 9.3.5 GetNewTarget ()
 - 9.3.6 GetGlobalObject ()
 - 9.4 Jobs and Host Operations to Enqueue Jobs
 - 9.4.1 JobCallback Records
 - 9.4.2 HostMakeJobCallback (*callback*)
 - 9.4.3 HostCallJobCallback (*jobCallback, V, argumentsList*)
 - 9.4.4 HostEnqueuePromiseJob (*job, realm*)
 - 9.5 InitializeHostDefinedRealm ()
 - 9.6 Agents
 - 9.6.1 AgentSignifier ()
 - 9.6.2 AgentCanSuspend ()
 - 9.7 Agent Clusters
 - 9.8 Forward Progress
 - 9.9 Processing Model of WeakRef and FinalizationRegistry Objects
 - 9.9.1 Objectives
 - 9.9.2 Liveness
 - 9.9.3 Execution
 - 9.9.4 Host Hooks
 - 9.9.4.1 HostEnqueueFinalizationRegistryCleanupJob (*finalizationRegistry*)
 - 9.10 ClearKeptObjects ()
 - 9.11 AddToKeptObjects (*object*)
 - 9.12 CleanupFinalizationRegistry (*finalizationRegistry*)
- ## 10 Ordinary and Exotic Objects Behaviours
- 10.1 Ordinary Object Internal Methods and Internal Slots
 - 10.1.1 [[GetPrototypeOf]] ()
 - 10.1.1.1 OrdinaryGetPrototypeOf (*O*)
 - 10.1.2 [[SetPrototypeOf]] (*V*)
 - 10.1.2.1 OrdinarySetPrototypeOf (*O, V*)
 - 10.1.3 [[IsExtensible]] ()
 - 10.1.3.1 OrdinaryIsExtensible (*O*)

- 10.1.4 `[[PreventExtensions]]` ()
 - 10.1.4.1 `OrdinaryPreventExtensions` (*O*)
 - 10.1.5 `[[GetOwnProperty]]` (*P*)
 - 10.1.5.1 `OrdinaryGetOwnProperty` (*O, P*)
 - 10.1.6 `[[DefineOwnProperty]]` (*P, Desc*)
 - 10.1.6.1 `OrdinaryDefineOwnProperty` (*O, P, Desc*)
 - 10.1.6.2 `IsCompatiblePropertyDescriptor` (*Extensible, Desc, Current*)
 - 10.1.6.3 `ValidateAndApplyPropertyDescriptor` (*O, P, extensible, Desc, current*)
 - 10.1.7 `[[HasProperty]]` (*P*)
 - 10.1.7.1 `OrdinaryHasProperty` (*O, P*)
 - 10.1.8 `[[Get]]` (*P, Receiver*)
 - 10.1.8.1 `OrdinaryGet` (*O, P, Receiver*)
 - 10.1.9 `[[Set]]` (*P, V, Receiver*)
 - 10.1.9.1 `OrdinarySet` (*O, P, V, Receiver*)
 - 10.1.9.2 `OrdinarySetWithOwnDescriptor` (*O, P, V, Receiver, ownDesc*)
 - 10.1.10 `[[Delete]]` (*P*)
 - 10.1.10.1 `OrdinaryDelete` (*O, P*)
 - 10.1.11 `[[OwnPropertyKeys]]` ()
 - 10.1.11.1 `OrdinaryOwnPropertyKeys` (*O*)
 - 10.1.12 `OrdinaryObjectCreate` (*proto [, additionalInternalSlotsList]*)
 - 10.1.13 `OrdinaryCreateFromConstructor` (*constructor, intrinsicDefaultProto [, internalSlotsList]*)
 - 10.1.14 `GetPrototypeFromConstructor` (*constructor, intrinsicDefaultProto*)
 - 10.1.15 `RequireInternalSlot` (*O, internalSlot*)
- 10.2 ECMAScript Function Objects
- 10.2.1 `[[Call]]` (*thisArgument, argumentsList*)
 - 10.2.1.1 `PrepareForOrdinaryCall` (*F, newTarget*)
 - 10.2.1.2 `OrdinaryCallBindThis` (*F, calleeContext, thisArgument*)
 - 10.2.1.3 RS: `EvaluateBody`
 - 10.2.1.4 `OrdinaryCallEvaluateBody` (*F, argumentsList*)
 - 10.2.2 `[[Construct]]` (*argumentsList, newTarget*)
 - 10.2.3 `OrdinaryFunctionCreate` (*functionPrototype, sourceText, ParameterList, Body, thisMode, Scope*)
 - 10.2.4 `AddRestrictedFunctionProperties` (*F, realm*)
 - 10.2.4.1 `%ThrowTypeError%` ()
 - 10.2.5 `MakeConstructor` (*F [, writablePrototype [, prototype]]*)
 - 10.2.6 `MakeClassConstructor` (*F*)
 - 10.2.7 `MakeMethod` (*F, homeObject*)
 - 10.2.8 `SetFunctionName` (*F, name [, prefix]*)
 - 10.2.9 `SetFunctionLength` (*F, length*)
 - 10.2.10 `FunctionDeclarationInstantiation` (*func, argumentsList*)
- 10.3 Built-in Function Objects
- 10.3.1 `[[Call]]` (*thisArgument, argumentsList*)
 - 10.3.2 `[[Construct]]` (*argumentsList, newTarget*)
 - 10.3.3 `CreateBuiltinFunction` (*steps, length, name, internalSlotsList [, realm [, prototype [, prefix]]]*)
- 10.4 Built-in Exotic Object Internal Methods and Slots
- 10.4.1 Bound Function Exotic Objects
 - 10.4.1.1 `[[Call]]` (*thisArgument, argumentsList*)
 - 10.4.1.2 `[[Construct]]` (*argumentsList, newTarget*)
 - 10.4.1.3 `BoundFunctionCreate` (*targetFunction, boundThis, boundArgs*)

10.4.2 Array Exotic Objects

10.4.2.1 `[[DefineOwnProperty]]` (*P*, *Desc*)

10.4.2.2 `ArrayCreate` (*length* [, *proto*])

10.4.2.3 `ArraySpeciesCreate` (*originalArray*, *length*)

10.4.2.4 `ArraySetLength` (*A*, *Desc*)

10.4.3 String Exotic Objects

10.4.3.1 `[[GetOwnProperty]]` (*P*)

10.4.3.2 `[[DefineOwnProperty]]` (*P*, *Desc*)

10.4.3.3 `[[OwnPropertyKeys]]` ()

10.4.3.4 `StringCreate` (*value*, *prototype*)

10.4.3.5 `StringGetOwnProperty` (*S*, *P*)

10.4.4 Arguments Exotic Objects

10.4.4.1 `[[GetOwnProperty]]` (*P*)

10.4.4.2 `[[DefineOwnProperty]]` (*P*, *Desc*)

10.4.4.3 `[[Get]]` (*P*, *Receiver*)

10.4.4.4 `[[Set]]` (*P*, *V*, *Receiver*)

10.4.4.5 `[[Delete]]` (*P*)

10.4.4.6 `CreateUnmappedArgumentsObject` (*argumentsList*)

10.4.4.7 `CreateMappedArgumentsObject` (*func*, *formals*, *argumentsList*, *env*)

10.4.4.7.1 `MakeArgGetter` (*name*, *env*)

10.4.4.7.2 `MakeArgSetter` (*name*, *env*)

10.4.5 Integer-Indexed Exotic Objects

10.4.5.1 `[[GetOwnProperty]]` (*P*)

10.4.5.2 `[[HasProperty]]` (*P*)

10.4.5.3 `[[DefineOwnProperty]]` (*P*, *Desc*)

10.4.5.4 `[[Get]]` (*P*, *Receiver*)

10.4.5.5 `[[Set]]` (*P*, *V*, *Receiver*)

10.4.5.6 `[[Delete]]` (*P*)

10.4.5.7 `[[OwnPropertyKeys]]` ()

10.4.5.8 `IntegerIndexedObjectCreate` (*prototype*)

10.4.5.9 `IsValidIntegerIndex` (*O*, *index*)

10.4.5.10 `IntegerIndexedElementGet` (*O*, *index*)

10.4.5.11 `IntegerIndexedElementSet` (*O*, *index*, *value*)

10.4.6 Module Namespace Exotic Objects

10.4.6.1 `[[SetPrototypeOf]]` (*V*)

10.4.6.2 `[[IsExtensible]]` ()

10.4.6.3 `[[PreventExtensions]]` ()

10.4.6.4 `[[GetOwnProperty]]` (*P*)

10.4.6.5 `[[DefineOwnProperty]]` (*P*, *Desc*)

10.4.6.6 `[[HasProperty]]` (*P*)

10.4.6.7 `[[Get]]` (*P*, *Receiver*)

10.4.6.8 `[[Set]]` (*P*, *V*, *Receiver*)

10.4.6.9 `[[Delete]]` (*P*)

10.4.6.10 `[[OwnPropertyKeys]]` ()

10.4.6.11 `ModuleNamespaceCreate` (*module*, *exports*)

10.4.7 Immutable Prototype Exotic Objects

10.4.7.1 `[[SetPrototypeOf]]` (*V*)

10.4.7.2 `SetImmutablePrototype` (*O*, *V*)

10.5 Proxy Object Internal Methods and Internal Slots

- 10.5.1 `[[GetPrototypeOf]]` ()
- 10.5.2 `[[SetPrototypeOf]]` (*V*)
- 10.5.3 `[[IsExtensible]]` ()
- 10.5.4 `[[PreventExtensions]]` ()
- 10.5.5 `[[GetOwnProperty]]` (*P*)
- 10.5.6 `[[DefineOwnProperty]]` (*P*, *Desc*)
- 10.5.7 `[[HasProperty]]` (*P*)
- 10.5.8 `[[Get]]` (*P*, *Receiver*)
- 10.5.9 `[[Set]]` (*P*, *V*, *Receiver*)
- 10.5.10 `[[Delete]]` (*P*)
- 10.5.11 `[[OwnPropertyKeys]]` ()
- 10.5.12 `[[Call]]` (*thisArgument*, *argumentsList*)
- 10.5.13 `[[Construct]]` (*argumentsList*, *newTarget*)
- 10.5.14 `ProxyCreate` (*target*, *handler*)

11 ECMAScript Language: Source Code

11.1 Source Text

- 11.1.1 SS: `UTF16EncodeCodePoint` (*cp*)
- 11.1.2 SS: `CodePointsToString` (*text*)
- 11.1.3 SS: `UTF16SurrogatePairToCodePoint` (*lead*, *trail*)
- 11.1.4 SS: `CodePointAt` (*string*, *position*)
- 11.1.5 SS: `StringToCodePoints` (*string*)
- 11.1.6 SS: `ParseText` (*sourceText*, *goalSymbol*)

11.2 Types of Source Code

- 11.2.1 Directive Prologues and the Use Strict Directive
- 11.2.2 Strict Mode Code
- 11.2.3 Non-ECMAScript Functions

12 ECMAScript Language: Lexical Grammar

- 12.1 Unicode Format-Control Characters
- 12.2 White Space
- 12.3 Line Terminators
- 12.4 Comments
- 12.5 Tokens
- 12.6 Names and Keywords
 - 12.6.1 Identifier Names
 - 12.6.1.1 SS: Early Errors
 - 12.6.2 Keywords and Reserved Words
- 12.7 Punctuators
- 12.8 Literals
 - 12.8.1 Null Literals
 - 12.8.2 Boolean Literals
 - 12.8.3 Numeric Literals
 - 12.8.3.1 SS: MV
 - 12.8.3.2 SS: NumericValue
 - 12.8.4 String Literals
 - 12.8.4.1 SS: SV
 - 12.8.4.2 SS: MV
 - 12.8.5 Regular Expression Literals

- 12.8.5.1 SS: Early Errors
- 12.8.5.2 SS: BodyText
- 12.8.5.3 SS: FlagText
- 12.8.6 Template Literal Lexical Components
 - 12.8.6.1 SS: TV and TRV
- 12.9 Automatic Semicolon Insertion
 - 12.9.1 Rules of Automatic Semicolon Insertion
 - 12.9.2 Examples of Automatic Semicolon Insertion
 - 12.9.3 Interesting Cases of Automatic Semicolon Insertion
 - 12.9.3.1 Interesting Cases of Automatic Semicolon Insertion in Statement Lists
 - 12.9.3.2 Cases of Automatic Semicolon Insertion and “[no *LineTerminator* here]”
 - 12.9.3.2.1 List of Grammar Productions with Optional Operands and “[no *LineTerminator* here]”
- 13 ECMAScript Language: Expressions
 - 13.1 Identifiers
 - 13.1.1 SS: Early Errors
 - 13.1.2 SS: StringValue
 - 13.1.3 RS: Evaluation
 - 13.2 Primary Expression
 - 13.2.1 Semantics
 - 13.2.1.1 SS: CoveredParenthesizedExpression
 - 13.2.2 The **this** Keyword
 - 13.2.2.1 RS: Evaluation
 - 13.2.3 Identifier Reference
 - 13.2.4 Literals
 - 13.2.4.1 RS: Evaluation
 - 13.2.5 Array Initializer
 - 13.2.5.1 RS: ArrayAccumulation
 - 13.2.5.2 RS: Evaluation
 - 13.2.6 Object Initializer
 - 13.2.6.1 SS: Early Errors
 - 13.2.6.2 SS: IsComputedPropertyKey
 - 13.2.6.3 SS: PropertyNameList
 - 13.2.6.4 RS: Evaluation
 - 13.2.6.5 RS: PropertyDefinitionEvaluation
 - 13.2.7 Function Defining Expressions
 - 13.2.8 Regular Expression Literals
 - 13.2.8.1 SS: Early Errors
 - 13.2.8.2 SS: IsValidRegularExpressionLiteral (*literal*)
 - 13.2.8.3 RS: Evaluation
 - 13.2.9 Template Literals
 - 13.2.9.1 SS: Early Errors
 - 13.2.9.2 SS: TemplateStrings
 - 13.2.9.3 GetTemplateObject (*templateLiteral*)
 - 13.2.9.4 RS: SubstitutionEvaluation
 - 13.2.9.5 RS: Evaluation
 - 13.2.10 The Grouping Operator
 - 13.2.10.1 SS: Early Errors
 - 13.2.10.2 RS: Evaluation

13.3 Left-Hand-Side Expressions

13.3.1 Static Semantics

13.3.1.1 SS: Early Errors

13.3.1.2 SS: CoveredCallExpression

13.3.2 Property Accessors

13.3.2.1 RS: Evaluation

13.3.3 EvaluatePropertyAccessWithExpressionKey (*baseValue, expression, strict*)

13.3.4 EvaluatePropertyAccessWithIdentifierKey (*baseValue, identifierName, strict*)

13.3.5 The **new** Operator

13.3.5.1 RS: Evaluation

13.3.5.1.1 EvaluateNew (*constructExpr, arguments*)

13.3.6 Function Calls

13.3.6.1 RS: Evaluation

13.3.6.2 EvaluateCall (*func, ref, arguments, tailPosition*)

13.3.7 The **super** Keyword

13.3.7.1 RS: Evaluation

13.3.7.2 GetSuperConstructor ()

13.3.7.3 MakeSuperPropertyReference (*actualThis, propertyKey, strict*)

13.3.8 Argument Lists

13.3.8.1 RS: ArgumentListEvaluation

13.3.9 Optional Chains

13.3.9.1 RS: Evaluation

13.3.9.2 RS: ChainEvaluation

13.3.10 Import Calls

13.3.10.1 RS: Evaluation

13.3.11 Tagged Templates

13.3.11.1 RS: Evaluation

13.3.12 Meta Properties

13.3.12.1 RS: Evaluation

13.3.12.1.1 HostGetImportMetaProperties (*moduleRecord*)

13.3.12.1.2 HostFinalizeImportMeta (*importMeta, moduleRecord*)

13.4 Update Expressions

13.4.1 SS: Early Errors

13.4.2 Postfix Increment Operator

13.4.2.1 RS: Evaluation

13.4.3 Postfix Decrement Operator

13.4.3.1 RS: Evaluation

13.4.4 Prefix Increment Operator

13.4.4.1 RS: Evaluation

13.4.5 Prefix Decrement Operator

13.4.5.1 RS: Evaluation

13.5 Unary Operators

13.5.1 The **delete** Operator

13.5.1.1 SS: Early Errors

13.5.1.2 RS: Evaluation

13.5.2 The **void** Operator

13.5.2.1 RS: Evaluation

13.5.3 The **typeof** Operator

- 13.5.3.1 RS: Evaluation
- 13.5.4 Unary **+** Operator
 - 13.5.4.1 RS: Evaluation
- 13.5.5 Unary **-** Operator
 - 13.5.5.1 RS: Evaluation
- 13.5.6 Bitwise NOT Operator (**~**)
 - 13.5.6.1 RS: Evaluation
- 13.5.7 Logical NOT Operator (**!**)
 - 13.5.7.1 RS: Evaluation
- 13.6 Exponentiation Operator
 - 13.6.1 RS: Evaluation
- 13.7 Multiplicative Operators
 - 13.7.1 RS: Evaluation
- 13.8 Additive Operators
 - 13.8.1 The Addition Operator (**+**)
 - 13.8.1.1 RS: Evaluation
 - 13.8.2 The Subtraction Operator (**-**)
 - 13.8.2.1 RS: Evaluation
- 13.9 Bitwise Shift Operators
 - 13.9.1 The Left Shift Operator (**<<**)
 - 13.9.1.1 RS: Evaluation
 - 13.9.2 The Signed Right Shift Operator (**>>**)
 - 13.9.2.1 RS: Evaluation
 - 13.9.3 The Unsigned Right Shift Operator (**>>>**)
 - 13.9.3.1 RS: Evaluation
- 13.10 Relational Operators
 - 13.10.1 RS: Evaluation
 - 13.10.2 InstanceofOperator (*V, target*)
- 13.11 Equality Operators
 - 13.11.1 RS: Evaluation
- 13.12 Binary Bitwise Operators
 - 13.12.1 RS: Evaluation
- 13.13 Binary Logical Operators
 - 13.13.1 RS: Evaluation
- 13.14 Conditional Operator (**?** **:**)
 - 13.14.1 RS: Evaluation
- 13.15 Assignment Operators
 - 13.15.1 SS: Early Errors
 - 13.15.2 RS: Evaluation
 - 13.15.3 ApplyStringOrNumericBinaryOperator (*lval, opText, rval*)
 - 13.15.4 EvaluateStringOrNumericBinaryExpression (*leftOperand, opText, rightOperand*)
 - 13.15.5 Destructuring Assignment
 - 13.15.5.1 SS: Early Errors
 - 13.15.5.2 RS: DestructuringAssignmentEvaluation
 - 13.15.5.3 RS: PropertyDestructuringAssignmentEvaluation
 - 13.15.5.4 RS: RestDestructuringAssignmentEvaluation
 - 13.15.5.5 RS: IteratorDestructuringAssignmentEvaluation

- 13.15.5.6 RS: KeyedDestructuringAssignmentEvaluation
- 13.16 Comma Operator (,)
 - 13.16.1 RS: Evaluation
- 14 ECMAScript Language: Statements and Declarations
 - 14.1 Statement Semantics
 - 14.1.1 RS: Evaluation
 - 14.2 Block
 - 14.2.1 SS: Early Errors
 - 14.2.2 RS: Evaluation
 - 14.2.3 BlockDeclarationInstantiation (*code, env*)
 - 14.3 Declarations and the Variable Statement
 - 14.3.1 Let and Const Declarations
 - 14.3.1.1 SS: Early Errors
 - 14.3.1.2 RS: Evaluation
 - 14.3.2 Variable Statement
 - 14.3.2.1 RS: Evaluation
 - 14.3.3 Destructuring Binding Patterns
 - 14.3.3.1 RS: PropertyBindingInitialization
 - 14.3.3.2 RS: RestBindingInitialization
 - 14.3.3.3 RS: KeyedBindingInitialization
 - 14.4 Empty Statement
 - 14.4.1 RS: Evaluation
 - 14.5 Expression Statement
 - 14.5.1 RS: Evaluation
 - 14.6 The **if** Statement
 - 14.6.1 SS: Early Errors
 - 14.6.2 RS: Evaluation
 - 14.7 Iteration Statements
 - 14.7.1 Semantics
 - 14.7.1.1 LoopContinues (*completion, labelSet*)
 - 14.7.1.2 RS: LoopEvaluation
 - 14.7.2 The **do-while** Statement
 - 14.7.2.1 SS: Early Errors
 - 14.7.2.2 RS: DoWhileLoopEvaluation
 - 14.7.3 The **while** Statement
 - 14.7.3.1 SS: Early Errors
 - 14.7.3.2 RS: WhileLoopEvaluation
 - 14.7.4 The **for** Statement
 - 14.7.4.1 SS: Early Errors
 - 14.7.4.2 RS: ForLoopEvaluation
 - 14.7.4.3 ForBodyEvaluation (*test, increment, stmt, perIterationBindings, labelSet*)
 - 14.7.4.4 CreatePerIterationEnvironment (*perIterationBindings*)
 - 14.7.5 The **for-in**, **for-of**, and **for-await-of** Statements
 - 14.7.5.1 SS: Early Errors
 - 14.7.5.2 SS: IsDestructuring
 - 14.7.5.3 RS: ForDeclarationBindingInitialization
 - 14.7.5.4 RS: ForDeclarationBindingInstantiation

- 14.7.5.5 RS: ForInOfLoopEvaluation
- 14.7.5.6 ForIn/OfHeadEvaluation (*uninitializedBoundNames, expr, iterationKind*)
- 14.7.5.7 ForIn/OfBodyEvaluation (*lhs, stmt, iteratorRecord, iterationKind, lhsKind, labelSet [, iteratorKind]*)
- 14.7.5.8 RS: Evaluation
- 14.7.5.9 EnumerateObjectProperties (*O*)
- 14.7.5.10 For-In Iterator Objects
 - 14.7.5.10.1 CreateForInIterator (*object*)
 - 14.7.5.10.2 The %ForInIteratorPrototype% Object
 - 14.7.5.10.2.1 %ForInIteratorPrototype%.next ()
 - 14.7.5.10.3 Properties of For-In Iterator Instances
- 14.8 The **continue** Statement
 - 14.8.1 SS: Early Errors
 - 14.8.2 RS: Evaluation
- 14.9 The **break** Statement
 - 14.9.1 SS: Early Errors
 - 14.9.2 RS: Evaluation
- 14.10 The **return** Statement
 - 14.10.1 RS: Evaluation
- 14.11 The **with** Statement
 - 14.11.1 SS: Early Errors
 - 14.11.2 RS: Evaluation
- 14.12 The **switch** Statement
 - 14.12.1 SS: Early Errors
 - 14.12.2 RS: CaseBlockEvaluation
 - 14.12.3 CaseClauseIsSelected (*C, input*)
 - 14.12.4 RS: Evaluation
- 14.13 Labelled Statements
 - 14.13.1 SS: Early Errors
 - 14.13.2 SS: IsLabelledFunction (*stmt*)
 - 14.13.3 RS: Evaluation
 - 14.13.4 RS: LabelledEvaluation
- 14.14 The **throw** Statement
 - 14.14.1 RS: Evaluation
- 14.15 The **try** Statement
 - 14.15.1 SS: Early Errors
 - 14.15.2 RS: CatchClauseEvaluation
 - 14.15.3 RS: Evaluation
- 14.16 The **debugger** Statement
 - 14.16.1 RS: Evaluation
- 15 ECMAScript Language: Functions and Classes
 - 15.1 Parameter Lists
 - 15.1.1 SS: Early Errors
 - 15.1.2 SS: ContainsExpression
 - 15.1.3 SS: IsSimpleParameterList
 - 15.1.4 SS: HasInitializer
 - 15.1.5 SS: ExpectedArgumentCount
 - 15.2 Function Definitions

- 15.2.1 SS: Early Errors
- 15.2.2 SS: FunctionBodyContainsUseStrict
- 15.2.3 RS: EvaluateFunctionBody
- 15.2.4 RS: InstantiateOrdinaryFunctionObject
- 15.2.5 RS: InstantiateOrdinaryFunctionExpression
- 15.2.6 RS: Evaluation
- 15.3 Arrow Function Definitions
 - 15.3.1 SS: Early Errors
 - 15.3.2 SS: ConciseBodyContainsUseStrict
 - 15.3.3 SS: CoveredFormalsList
 - 15.3.4 RS: EvaluateConciseBody
 - 15.3.5 RS: InstantiateArrowFunctionExpression
 - 15.3.6 RS: Evaluation
- 15.4 Method Definitions
 - 15.4.1 SS: Early Errors
 - 15.4.2 SS: HasDirectSuper
 - 15.4.3 SS: SpecialMethod
 - 15.4.4 RS: DefineMethod
 - 15.4.5 RS: MethodDefinitionEvaluation
- 15.5 Generator Function Definitions
 - 15.5.1 SS: Early Errors
 - 15.5.2 RS: EvaluateGeneratorBody
 - 15.5.3 RS: InstantiateGeneratorFunctionObject
 - 15.5.4 RS: InstantiateGeneratorFunctionExpression
 - 15.5.5 RS: Evaluation
- 15.6 Async Generator Function Definitions
 - 15.6.1 SS: Early Errors
 - 15.6.2 RS: EvaluateAsyncGeneratorBody
 - 15.6.3 RS: InstantiateAsyncGeneratorFunctionObject
 - 15.6.4 RS: InstantiateAsyncGeneratorFunctionExpression
 - 15.6.5 RS: Evaluation
- 15.7 Class Definitions
 - 15.7.1 SS: Early Errors
 - 15.7.2 SS: ClassElementKind
 - 15.7.3 SS: ConstructorMethod
 - 15.7.4 SS: IsStatic
 - 15.7.5 SS: NonConstructorMethodDefinitions
 - 15.7.6 SS: PrototypePropertyNameList
 - 15.7.7 RS: ClassDefinitionEvaluation
 - 15.7.7.1 Default Constructor Functions
 - 15.7.8 RS: BindingClassDeclarationEvaluation
 - 15.7.9 RS: Evaluation
- 15.8 Async Function Definitions
 - 15.8.1 SS: Early Errors
 - 15.8.2 RS: InstantiateAsyncFunctionObject
 - 15.8.3 RS: InstantiateAsyncFunctionExpression
 - 15.8.4 RS: EvaluateAsyncFunctionBody
 - 15.8.5 RS: Evaluation

- 15.9 Async Arrow Function Definitions
 - 15.9.1 SS: Early Errors
 - 15.9.2 SS: CoveredAsyncArrowHead
 - 15.9.3 SS: AsyncConciseBodyContainsUseStrict
 - 15.9.4 RS: EvaluateAsyncConciseBody
 - 15.9.5 RS: InstantiateAsyncArrowFunctionExpression
 - 15.9.6 RS: Evaluation
- 15.10 Tail Position Calls
 - 15.10.1 SS: IsInTailPosition (*call*)
 - 15.10.2 SS: HasCallInTailPosition
 - 15.10.2.1 Statement Rules
 - 15.10.2.2 Expression Rules
 - 15.10.3 PrepareForTailCall ()
- 16 ECMAScript Language: Scripts and Modules
 - 16.1 Scripts
 - 16.1.1 SS: Early Errors
 - 16.1.2 SS: IsStrict
 - 16.1.3 RS: Evaluation
 - 16.1.4 Script Records
 - 16.1.5 ParseScript (*sourceText, realm, hostDefined*)
 - 16.1.6 ScriptEvaluation (*scriptRecord*)
 - 16.1.7 GlobalDeclarationInstantiation (*script, env*)
 - 16.2 Modules
 - 16.2.1 Module Semantics
 - 16.2.1.1 SS: Early Errors
 - 16.2.1.2 SS: ImportedLocalNames (*importEntries*)
 - 16.2.1.3 SS: ModuleRequests
 - 16.2.1.4 Abstract Module Records
 - 16.2.1.5 Cyclic Module Records
 - 16.2.1.5.1 Link () Concrete Method
 - 16.2.1.5.1.1 InnerModuleLinking (*module, stack, index*)
 - 16.2.1.5.2 Evaluate () Concrete Method
 - 16.2.1.5.2.1 InnerModuleEvaluation (*module, stack, index*)
 - 16.2.1.5.3 Example Cyclic Module Record Graphs
 - 16.2.1.6 Source Text Module Records
 - 16.2.1.6.1 ParseModule (*sourceText, realm, hostDefined*)
 - 16.2.1.6.2 GetExportedNames ([*exportStarSet*]) Concrete Method
 - 16.2.1.6.3 ResolveExport (*exportName* [, *resolveSet*]) Concrete Method
 - 16.2.1.6.4 InitializeEnvironment () Concrete Method
 - 16.2.1.6.5 ExecuteModule () Concrete Method
 - 16.2.1.7 HostResolveImportedModule (*referencingScriptOrModule, specifier*)
 - 16.2.1.8 HostImportModuleDynamically (*referencingScriptOrModule, specifier, promiseCapability*)
 - 16.2.1.9 FinishDynamicImport (*referencingScriptOrModule, specifier, promiseCapability, completion*)
 - 16.2.1.10 GetModuleNamespace (*module*)
 - 16.2.1.11 RS: Evaluation
 - 16.2.2 Imports
 - 16.2.2.1 SS: Early Errors
 - 16.2.2.2 SS: ImportEntries

- 16.2.2.3 SS: ImportEntriesForModule
- 16.2.3 Exports
 - 16.2.3.1 SS: Early Errors
 - 16.2.3.2 SS: ExportedBindings
 - 16.2.3.3 SS: ExportedNames
 - 16.2.3.4 SS: ExportEntries
 - 16.2.3.5 SS: ExportEntriesForModule
 - 16.2.3.6 SS: ReferencedBindings
 - 16.2.3.7 RS: Evaluation
- 17 Error Handling and Language Extensions
- 17.1 Forbidden Extensions
- 18 ECMAScript Standard Built-in Objects
- 19 The Global Object
 - 19.1 Value Properties of the Global Object
 - 19.1.1 globalThis
 - 19.1.2 Infinity
 - 19.1.3 NaN
 - 19.1.4 undefined
 - 19.2 Function Properties of the Global Object
 - 19.2.1 eval (*x*)
 - 19.2.1.1 PerformEval (*x*, *callerRealm*, *strictCaller*, *direct*)
 - 19.2.1.2 HostEnsureCanCompileStrings (*callerRealm*, *calleeRealm*)
 - 19.2.1.3 EvalDeclarationInstantiation (*body*, *varEnv*, *lexEnv*, *strict*)
 - 19.2.2 isFinite (*number*)
 - 19.2.3 isNaN (*number*)
 - 19.2.4 parseFloat (*string*)
 - 19.2.5 parseInt (*string*, *radix*)
 - 19.2.6 URI Handling Functions
 - 19.2.6.1 URI Syntax and Semantics
 - 19.2.6.1.1 Encode (*string*, *unescapedSet*)
 - 19.2.6.1.2 Decode (*string*, *reservedSet*)
 - 19.2.6.2 decodeURI (*encodedURI*)
 - 19.2.6.3 decodeURIComponent (*encodedURIComponent*)
 - 19.2.6.4 encodeURI (*uri*)
 - 19.2.6.5 encodeURIComponent (*uriComponent*)
 - 19.3 Constructor Properties of the Global Object
 - 19.3.1 Array (. . .)
 - 19.3.2 ArrayBuffer (. . .)
 - 19.3.3 BigInt (. . .)
 - 19.3.4 BigInt64Array (. . .)
 - 19.3.5 BigUint64Array (. . .)
 - 19.3.6 Boolean (. . .)
 - 19.3.7 DataView (. . .)
 - 19.3.8 Date (. . .)
 - 19.3.9 Error (. . .)
 - 19.3.10 EvalError (. . .)
 - 19.3.11 FinalizationRegistry (. . .)
 - 19.3.12 Float32Array (. . .)

- 19.3.13 Float64Array (...)
- 19.3.14 Function (...)
- 19.3.15 Int8Array (...)
- 19.3.16 Int16Array (...)
- 19.3.17 Int32Array (...)
- 19.3.18 Map (...)
- 19.3.19 Number (...)
- 19.3.20 Object (...)
- 19.3.21 Promise (...)
- 19.3.22 Proxy (...)
- 19.3.23 RangeError (...)
- 19.3.24 ReferenceError (...)
- 19.3.25 RegExp (...)
- 19.3.26 Set (...)
- 19.3.27 SharedArrayBuffer (...)
- 19.3.28 String (...)
- 19.3.29 Symbol (...)
- 19.3.30 SyntaxError (...)
- 19.3.31 TypeError (...)
- 19.3.32 Uint8Array (...)
- 19.3.33 Uint8ClampedArray (...)
- 19.3.34 Uint16Array (...)
- 19.3.35 Uint32Array (...)
- 19.3.36 URIError (...)
- 19.3.37 WeakMap (...)
- 19.3.38 WeakRef (...)
- 19.3.39 WeakSet (...)

19.4 Other Properties of the Global Object

- 19.4.1 Atomics
- 19.4.2 JSON
- 19.4.3 Math
- 19.4.4 Reflect

20 Fundamental Objects

20.1 Object Objects

20.1.1 The Object Constructor

- 20.1.1.1 Object ([*value*])

20.1.2 Properties of the Object Constructor

- 20.1.2.1 Object.assign (*target*, ...*sources*)
- 20.1.2.2 Object.create (*O*, *Properties*)
- 20.1.2.3 Object.defineProperties (*O*, *Properties*)
 - 20.1.2.3.1 ObjectDefineProperty (*O*, *Properties*)
- 20.1.2.4 Object.defineProperty (*O*, *P*, *Attributes*)
- 20.1.2.5 Object.entries (*O*)
- 20.1.2.6 Object.freeze (*O*)
- 20.1.2.7 Object.fromEntries (*iterable*)

- 20.1.2.7.1 CreateDataPropertyOnObject Functions

- 20.1.2.8 Object.getOwnPropertyDescriptor (*O*, *P*)
- 20.1.2.9 Object.getOwnPropertyDescriptors (*O*)

- 20.1.2.10 Object.getOwnPropertyNames (*O*)
- 20.1.2.11 Object.getOwnPropertySymbols (*O*)
 - 20.1.2.11.1 GetOwnPropertyKeys (*O*, *type*)
- 20.1.2.12 Object.getPrototypeOf (*O*)
- 20.1.2.13 Object.is (*value1*, *value2*)
- 20.1.2.14 Object.isExtensible (*O*)
- 20.1.2.15 Object.isFrozen (*O*)
- 20.1.2.16 Object.isSealed (*O*)
- 20.1.2.17 Object.keys (*O*)
- 20.1.2.18 Object.preventExtensions (*O*)
- 20.1.2.19 Object.prototype
- 20.1.2.20 Object.seal (*O*)
- 20.1.2.21 Object.setPrototypeOf (*O*, *proto*)
- 20.1.2.22 Object.values (*O*)
- 20.1.3 Properties of the Object Prototype Object
 - 20.1.3.1 Object.prototype.constructor
 - 20.1.3.2 Object.prototype.hasOwnProperty (*V*)
 - 20.1.3.3 Object.prototype.isPrototypeOf (*V*)
 - 20.1.3.4 Object.prototype.propertyIsEnumerable (*V*)
 - 20.1.3.5 Object.prototype.toLocaleString ([*reserved1* [, *reserved2*]])
 - 20.1.3.6 Object.prototype.toString ()
 - 20.1.3.7 Object.prototype.valueOf ()
- 20.1.4 Properties of Object Instances
- 20.2 Function Objects
 - 20.2.1 The Function Constructor
 - 20.2.1.1 Function (*p1*, *p2*, ... , *pn*, *body*)
 - 20.2.1.1.1 CreateDynamicFunction (*constructor*, *newTarget*, *kind*, *args*)
 - 20.2.2 Properties of the Function Constructor
 - 20.2.2.1 Function.length
 - 20.2.2.2 Function.prototype
 - 20.2.3 Properties of the Function Prototype Object
 - 20.2.3.1 Function.prototype.apply (*thisArg*, *argArray*)
 - 20.2.3.2 Function.prototype.bind (*thisArg*, ...*args*)
 - 20.2.3.3 Function.prototype.call (*thisArg*, ...*args*)
 - 20.2.3.4 Function.prototype.constructor
 - 20.2.3.5 Function.prototype.toString ()
 - 20.2.3.6 Function.prototype [@@hasInstance] (*V*)
 - 20.2.4 Function Instances
 - 20.2.4.1 length
 - 20.2.4.2 name
 - 20.2.4.3 prototype
 - 20.2.5 HostHasSourceTextAvailable (*func*)
- 20.3 Boolean Objects
 - 20.3.1 The Boolean Constructor
 - 20.3.1.1 Boolean (*value*)
 - 20.3.2 Properties of the Boolean Constructor
 - 20.3.2.1 Boolean.prototype
 - 20.3.3 Properties of the Boolean Prototype Object

- 20.3.3.1 Boolean.prototype.constructor
- 20.3.3.2 Boolean.prototype.toString ()
- 20.3.3.3 Boolean.prototype.valueOf ()
- 20.3.4 Properties of Boolean Instances
- 20.4 Symbol Objects
 - 20.4.1 The Symbol Constructor
 - 20.4.1.1 Symbol ([*description*])
 - 20.4.2 Properties of the Symbol Constructor
 - 20.4.2.1 Symbol.asyncIterator
 - 20.4.2.2 Symbol.for (*key*)
 - 20.4.2.3 Symbol.hasInstance
 - 20.4.2.4 Symbol.isConcatSpreadable
 - 20.4.2.5 Symbol.iterator
 - 20.4.2.6 Symbol.keyFor (*sym*)
 - 20.4.2.7 Symbol.match
 - 20.4.2.8 Symbol.matchAll
 - 20.4.2.9 Symbol.prototype
 - 20.4.2.10 Symbol.replace
 - 20.4.2.11 Symbol.search
 - 20.4.2.12 Symbol.species
 - 20.4.2.13 Symbol.split
 - 20.4.2.14 Symbol.toPrimitive
 - 20.4.2.15 Symbol.toStringTag
 - 20.4.2.16 Symbol.unscopables
 - 20.4.3 Properties of the Symbol Prototype Object
 - 20.4.3.1 Symbol.prototype.constructor
 - 20.4.3.2 get Symbol.prototype.description
 - 20.4.3.3 Symbol.prototype.toString ()
 - 20.4.3.3.1 SymbolDescriptiveString (*sym*)
 - 20.4.3.4 Symbol.prototype.valueOf ()
 - 20.4.3.5 Symbol.prototype [@@toPrimitive] (*hint*)
 - 20.4.3.6 Symbol.prototype [@@toStringTag]
 - 20.4.4 Properties of Symbol Instances
- 20.5 Error Objects
 - 20.5.1 The Error Constructor
 - 20.5.1.1 Error (*message*)
 - 20.5.2 Properties of the Error Constructor
 - 20.5.2.1 Error.prototype
 - 20.5.3 Properties of the Error Prototype Object
 - 20.5.3.1 Error.prototype.constructor
 - 20.5.3.2 Error.prototype.message
 - 20.5.3.3 Error.prototype.name
 - 20.5.3.4 Error.prototype.toString ()
 - 20.5.4 Properties of Error Instances
 - 20.5.5 Native Error Types Used in This Standard
 - 20.5.5.1 EvalError
 - 20.5.5.2 RangeError
 - 20.5.5.3 ReferenceError

- 20.5.5.4 SyntaxError
- 20.5.5.5 TypeError
- 20.5.5.6 URIError
- 20.5.6 *NativeError* Object Structure
 - 20.5.6.1 The *NativeError* Constructors
 - 20.5.6.1.1 *NativeError* (*message*)
 - 20.5.6.2 Properties of the *NativeError* Constructors
 - 20.5.6.2.1 *NativeError*.prototype
 - 20.5.6.3 Properties of the *NativeError* Prototype Objects
 - 20.5.6.3.1 *NativeError*.prototype.constructor
 - 20.5.6.3.2 *NativeError*.prototype.message
 - 20.5.6.3.3 *NativeError*.prototype.name
 - 20.5.6.4 Properties of *NativeError* Instances
- 20.5.7 AggregateError Objects
 - 20.5.7.1 The AggregateError Constructor
 - 20.5.7.1.1 AggregateError (*errors*, *message*)
 - 20.5.7.2 Properties of the AggregateError Constructor
 - 20.5.7.2.1 AggregateError.prototype
 - 20.5.7.3 Properties of the AggregateError Prototype Object
 - 20.5.7.3.1 AggregateError.prototype.constructor
 - 20.5.7.3.2 AggregateError.prototype.message
 - 20.5.7.3.3 AggregateError.prototype.name
 - 20.5.7.4 Properties of AggregateError Instances
- 21 Numbers and Dates
 - 21.1 Number Objects
 - 21.1.1 The Number Constructor
 - 21.1.1.1 Number (*value*)
 - 21.1.2 Properties of the Number Constructor
 - 21.1.2.1 Number.EPSILON
 - 21.1.2.2 Number.isFinite (*number*)
 - 21.1.2.3 Number.isInteger (*number*)
 - 21.1.2.4 Number.isNaN (*number*)
 - 21.1.2.5 Number.isSafeInteger (*number*)
 - 21.1.2.6 Number.MAX_SAFE_INTEGER
 - 21.1.2.7 Number.MAX_VALUE
 - 21.1.2.8 Number.MIN_SAFE_INTEGER
 - 21.1.2.9 Number.MIN_VALUE
 - 21.1.2.10 Number.NaN
 - 21.1.2.11 Number.NEGATIVE_INFINITY
 - 21.1.2.12 Number.parseFloat (*string*)
 - 21.1.2.13 Number.parseInt (*string*, *radix*)
 - 21.1.2.14 Number.POSITIVE_INFINITY
 - 21.1.2.15 Number.prototype
 - 21.1.3 Properties of the Number Prototype Object
 - 21.1.3.1 Number.prototype.constructor
 - 21.1.3.2 Number.prototype.toExponential (*fractionDigits*)
 - 21.1.3.3 Number.prototype.toFixed (*fractionDigits*)
 - 21.1.3.4 Number.prototype.toLocaleString ([*reserved1* [, *reserved2*]])

- 21.1.3.5 Number.prototype.toPrecision (*precision*)
- 21.1.3.6 Number.prototype.toString ([*radix*])
- 21.1.3.7 Number.prototype.valueOf ()
- 21.1.4 Properties of Number Instances
- 21.2 BigInt Objects
 - 21.2.1 The BigInt Constructor
 - 21.2.1.1 BigInt (*value*)
 - 21.2.1.1.1 NumberToBigInt (*number*)
 - 21.2.2 Properties of the BigInt Constructor
 - 21.2.2.1 BigInt.asIntN (*bits*, *bigint*)
 - 21.2.2.2 BigInt.asUintN (*bits*, *bigint*)
 - 21.2.2.3 BigInt.prototype
 - 21.2.3 Properties of the BigInt Prototype Object
 - 21.2.3.1 BigInt.prototype.constructor
 - 21.2.3.2 BigInt.prototype.toLocaleString ([*reserved1* [, *reserved2*]])
 - 21.2.3.3 BigInt.prototype.toString ([*radix*])
 - 21.2.3.4 BigInt.prototype.valueOf ()
 - 21.2.3.5 BigInt.prototype [@@toStringTag]
- 21.3 The Math Object
 - 21.3.1 Value Properties of the Math Object
 - 21.3.1.1 Math.E
 - 21.3.1.2 Math.LN10
 - 21.3.1.3 Math.LN2
 - 21.3.1.4 Math.LOG10E
 - 21.3.1.5 Math.LOG2E
 - 21.3.1.6 Math.PI
 - 21.3.1.7 Math.SQRT1_2
 - 21.3.1.8 Math.SQRT2
 - 21.3.1.9 Math [@@toStringTag]
 - 21.3.2 Function Properties of the Math Object
 - 21.3.2.1 Math.abs (*x*)
 - 21.3.2.2 Math.acos (*x*)
 - 21.3.2.3 Math.acosh (*x*)
 - 21.3.2.4 Math.asin (*x*)
 - 21.3.2.5 Math.asinh (*x*)
 - 21.3.2.6 Math.atan (*x*)
 - 21.3.2.7 Math.atanh (*x*)
 - 21.3.2.8 Math.atan2 (*y*, *x*)
 - 21.3.2.9 Math.cbrt (*x*)
 - 21.3.2.10 Math.ceil (*x*)
 - 21.3.2.11 Math.clz32 (*x*)
 - 21.3.2.12 Math.cos (*x*)
 - 21.3.2.13 Math.cosh (*x*)
 - 21.3.2.14 Math.exp (*x*)
 - 21.3.2.15 Math.expm1 (*x*)
 - 21.3.2.16 Math.floor (*x*)
 - 21.3.2.17 Math.fround (*x*)
 - 21.3.2.18 Math.hypot (...*args*)

- 21.3.2.19 `Math.imul (x, y)`
- 21.3.2.20 `Math.log (x)`
- 21.3.2.21 `Math.log1p (x)`
- 21.3.2.22 `Math.log10 (x)`
- 21.3.2.23 `Math.log2 (x)`
- 21.3.2.24 `Math.max (...args)`
- 21.3.2.25 `Math.min (...args)`
- 21.3.2.26 `Math.pow (base, exponent)`
- 21.3.2.27 `Math.random ()`
- 21.3.2.28 `Math.round (x)`
- 21.3.2.29 `Math.sign (x)`
- 21.3.2.30 `Math.sin (x)`
- 21.3.2.31 `Math.sinh (x)`
- 21.3.2.32 `Math.sqrt (x)`
- 21.3.2.33 `Math.tan (x)`
- 21.3.2.34 `Math.tanh (x)`
- 21.3.2.35 `Math.trunc (x)`

21.4 Date Objects

21.4.1 Overview of Date Objects and Definitions of Abstract Operations

- 21.4.1.1 Time Values and Time Range
- 21.4.1.2 Day Number and Time within Day
- 21.4.1.3 Year Number
- 21.4.1.4 Month Number
- 21.4.1.5 Date Number
- 21.4.1.6 Week Day
- 21.4.1.7 `LocalTZA (t, isUTC)`
- 21.4.1.8 `LocalTime (t)`
- 21.4.1.9 `UTC (t)`
- 21.4.1.10 Hours, Minutes, Second, and Milliseconds
- 21.4.1.11 `MakeTime (hour, min, sec, ms)`
- 21.4.1.12 `MakeDay (year, month, date)`
- 21.4.1.13 `MakeDate (day, time)`
- 21.4.1.14 `TimeClip (time)`
- 21.4.1.15 Date Time String Format

- 21.4.1.15.1 Expanded Years

21.4.2 The Date Constructor

- 21.4.2.1 `Date (...values)`

21.4.3 Properties of the Date Constructor

- 21.4.3.1 `Date.now ()`
- 21.4.3.2 `Date.parse (string)`
- 21.4.3.3 `Date.prototype`
- 21.4.3.4 `Date.UTC (year [, month [, date [, hours [, minutes [, seconds [, ms]]]]]]]]]])`

21.4.4 Properties of the Date Prototype Object

- 21.4.4.1 `Date.prototype.constructor`
- 21.4.4.2 `Date.prototype.getDate ()`
- 21.4.4.3 `Date.prototype.getDay ()`
- 21.4.4.4 `Date.prototype.getFullYear ()`
- 21.4.4.5 `Date.prototype.getHours ()`

- 21.4.4.6 Date.prototype.getMilliseconds ()
- 21.4.4.7 Date.prototype.getMinutes ()
- 21.4.4.8 Date.prototype.getMonth ()
- 21.4.4.9 Date.prototype.getSeconds ()
- 21.4.4.10 Date.prototype.getTime ()
- 21.4.4.11 Date.prototype.getTimezoneOffset ()
- 21.4.4.12 Date.prototype.getUTCDate ()
- 21.4.4.13 Date.prototype.getUTCDay ()
- 21.4.4.14 Date.prototype.getUTCFullYear ()
- 21.4.4.15 Date.prototype.getUTCHours ()
- 21.4.4.16 Date.prototype.getUTCMilliseconds ()
- 21.4.4.17 Date.prototype.getUTCMinutes ()
- 21.4.4.18 Date.prototype.getUTCMonth ()
- 21.4.4.19 Date.prototype.getUTCSeconds ()
- 21.4.4.20 Date.prototype.setDate (*date*)
- 21.4.4.21 Date.prototype.setFullYear (*year* [, *month* [, *date*]])
- 21.4.4.22 Date.prototype.setHours (*hour* [, *min* [, *sec* [, *ms*]]])
- 21.4.4.23 Date.prototype.setMilliseconds (*ms*)
- 21.4.4.24 Date.prototype.setMinutes (*min* [, *sec* [, *ms*]])
- 21.4.4.25 Date.prototype.setMonth (*month* [, *date*])
- 21.4.4.26 Date.prototype.setSeconds (*sec* [, *ms*])
- 21.4.4.27 Date.prototype.setTime (*time*)
- 21.4.4.28 Date.prototype.setUTCDate (*date*)
- 21.4.4.29 Date.prototype.setUTCFullYear (*year* [, *month* [, *date*]])
- 21.4.4.30 Date.prototype.setUTCHours (*hour* [, *min* [, *sec* [, *ms*]]])
- 21.4.4.31 Date.prototype.setUTCMilliseconds (*ms*)
- 21.4.4.32 Date.prototype.setUTCMinutes (*min* [, *sec* [, *ms*]])
- 21.4.4.33 Date.prototype.setUTCMonth (*month* [, *date*])
- 21.4.4.34 Date.prototype.setUTCSeconds (*sec* [, *ms*])
- 21.4.4.35 Date.prototype.toString ()
- 21.4.4.36 Date.prototype.toISOString ()
- 21.4.4.37 Date.prototype.toJSON (*key*)
- 21.4.4.38 Date.prototype.toLocaleDateString ([*reserved1* [, *reserved2*]])
- 21.4.4.39 Date.prototype.toLocaleString ([*reserved1* [, *reserved2*]])
- 21.4.4.40 Date.prototype.toLocaleTimeString ([*reserved1* [, *reserved2*]])
- 21.4.4.41 Date.prototype.toString ()
 - 21.4.4.41.1 TimeString (*tv*)
 - 21.4.4.41.2 DateString (*tv*)
 - 21.4.4.41.3 TimeZoneString (*tv*)
 - 21.4.4.41.4 ToDateString (*tv*)
- 21.4.4.42 Date.prototype.toTimeString ()
- 21.4.4.43 Date.prototype.toUTCString ()
- 21.4.4.44 Date.prototype.valueOf ()
- 21.4.4.45 Date.prototype [@@toPrimitive] (*hint*)

21.4.5 Properties of Date Instances

22 Text Processing

22.1 String Objects

22.1.1 The String Constructor

- 22.1.1.1 String (*value*)
- 22.1.2 Properties of the String Constructor
 - 22.1.2.1 String.fromCharCode (...*codeUnits*)
 - 22.1.2.2 String.fromCharCodePoint (...*codePoints*)
 - 22.1.2.3 String.prototype
 - 22.1.2.4 String.raw (*template*, ...*substitutions*)
- 22.1.3 Properties of the String Prototype Object
 - 22.1.3.1 String.prototype.charAt (*pos*)
 - 22.1.3.2 String.prototype.charCodeAt (*pos*)
 - 22.1.3.3 String.prototype.codePointAt (*pos*)
 - 22.1.3.4 String.prototype.concat (...*args*)
 - 22.1.3.5 String.prototype.constructor
 - 22.1.3.6 String.prototype.endsWith (*searchString* [, *endPosition*])
 - 22.1.3.7 String.prototype.includes (*searchString* [, *position*])
 - 22.1.3.8 String.prototype.indexOf (*searchString* [, *position*])
 - 22.1.3.9 String.prototype.lastIndexOf (*searchString* [, *position*])
 - 22.1.3.10 String.prototype.localeCompare (*that* [, *reserved1* [, *reserved2*]])
 - 22.1.3.11 String.prototype.match (*regexp*)
 - 22.1.3.12 String.prototype.matchAll (*regexp*)
 - 22.1.3.13 String.prototype.normalize ([*form*])
 - 22.1.3.14 String.prototype.padEnd (*maxLength* [, *fillString*])
 - 22.1.3.15 String.prototype.padStart (*maxLength* [, *fillString*])
 - 22.1.3.15.1 StringPad (*O*, *maxLength*, *fillString*, *placement*)
 - 22.1.3.16 String.prototype.repeat (*count*)
 - 22.1.3.17 String.prototype.replace (*searchValue*, *replaceValue*)
 - 22.1.3.17.1 GetSubstitution (*matched*, *str*, *position*, *captures*, *namedCaptures*, *replacement*)
 - 22.1.3.18 String.prototype.replaceAll (*searchValue*, *replaceValue*)
 - 22.1.3.19 String.prototype.search (*regexp*)
 - 22.1.3.20 String.prototype.slice (*start*, *end*)
 - 22.1.3.21 String.prototype.split (*separator*, *limit*)
 - 22.1.3.21.1 SplitMatch (*S*, *q*, *R*)
 - 22.1.3.22 String.prototype.startsWith (*searchString* [, *position*])
 - 22.1.3.23 String.prototype.substring (*start*, *end*)
 - 22.1.3.24 String.prototype.toLocaleLowerCase ([*reserved1* [, *reserved2*]])
 - 22.1.3.25 String.prototype.toLocaleUpperCase ([*reserved1* [, *reserved2*]])
 - 22.1.3.26 String.prototype.toLowerCase ()
 - 22.1.3.27 String.prototype.toString ()
 - 22.1.3.28 String.prototype.toUpperCase ()
 - 22.1.3.29 String.prototype.trim ()
 - 22.1.3.29.1 TrimString (*string*, *where*)
 - 22.1.3.30 String.prototype.trimEnd ()
 - 22.1.3.31 String.prototype.trimStart ()
 - 22.1.3.32 String.prototype.valueOf ()
 - 22.1.3.33 String.prototype [@@iterator] ()
- 22.1.4 Properties of String Instances
 - 22.1.4.1 length
- 22.1.5 String Iterator Objects
 - 22.1.5.1 The %StringIteratorPrototype% Object

22.1.5.1.1 %StringIteratorPrototype%.next ()

22.1.5.1.2 %StringIteratorPrototype% [@@toStringTag]

22.2 RegExp (Regular Expression) Objects

22.2.1 Patterns

22.2.1.1 SS: Early Errors

22.2.1.2 SS: CapturingGroupName

22.2.1.3 SS: IsCharacterClass

22.2.1.4 SS: CharacterValue

22.2.1.5 SS: SourceText

22.2.1.6 SS: CapturingGroupName

22.2.2 Pattern Semantics

22.2.2.1 Notation

22.2.2.2 Pattern

22.2.2.3 Disjunction

22.2.2.4 Alternative

22.2.2.5 Term

22.2.2.5.1 RepeatMatcher (*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*)

22.2.2.6 Assertion

22.2.2.6.1 IsWordChar (*e*)

22.2.2.7 Quantifier

22.2.2.8 Atom

22.2.2.8.1 CharacterSetMatcher (*A*, *invert*, *direction*)

22.2.2.8.2 Canonicalize (*ch*)

22.2.2.8.3 UnicodeMatchProperty (*p*)

22.2.2.8.4 UnicodeMatchPropertyValue (*p*, *v*)

22.2.2.9 AtomEscape

22.2.2.9.1 BackreferenceMatcher (*n*, *direction*)

22.2.2.10 CharacterEscape

22.2.2.11 DecimalEscape

22.2.2.12 CharacterClassEscape

22.2.2.13 CharacterClass

22.2.2.14 ClassRanges

22.2.2.15 NonemptyClassRanges

22.2.2.15.1 CharacterRange (*A*, *B*)

22.2.2.16 NonemptyClassRangesNoDash

22.2.2.17 ClassAtom

22.2.2.18 ClassAtomNoDash

22.2.2.19 ClassEscape

22.2.3 The RegExp Constructor

22.2.3.1 RegExp (*pattern*, *flags*)

22.2.3.2 Abstract Operations for the RegExp Constructor

22.2.3.2.1 RegExpAlloc (*newTarget*)

22.2.3.2.2 RegExpInitialize (*obj*, *pattern*, *flags*)

22.2.3.2.3 SS: ParsePattern (*patternText*, *u*)

22.2.3.2.4 RegExpCreate (*P*, *F*)

22.2.3.2.5 EscapeRegExpPattern (*P*, *F*)

22.2.4 Properties of the RegExp Constructor

22.2.4.1 RegExp.prototype

- 22.2.4.2 get RegExp [@@species]
- 22.2.5 Properties of the RegExp Prototype Object
 - 22.2.5.1 RegExp.prototype.constructor
 - 22.2.5.2 RegExp.prototype.exec (*string*)
 - 22.2.5.2.1 RegExpExec (*R*, *S*)
 - 22.2.5.2.2 RegExpBuiltinExec (*R*, *S*)
 - 22.2.5.2.3 AdvanceStringIndex (*S*, *index*, *unicode*)
 - 22.2.5.3 get RegExp.prototype.dotAll
 - 22.2.5.4 get RegExp.prototype.flags
 - 22.2.5.5 get RegExp.prototype.global
 - 22.2.5.6 get RegExp.prototype.ignoreCase
 - 22.2.5.7 RegExp.prototype [@@match] (*string*)
 - 22.2.5.8 RegExp.prototype [@@matchAll] (*string*)
 - 22.2.5.9 get RegExp.prototype.multiline
 - 22.2.5.10 RegExp.prototype [@@replace] (*string*, *replaceValue*)
 - 22.2.5.11 RegExp.prototype [@@search] (*string*)
 - 22.2.5.12 get RegExp.prototype.source
 - 22.2.5.13 RegExp.prototype [@@split] (*string*, *limit*)
 - 22.2.5.14 get RegExp.prototype.sticky
 - 22.2.5.15 RegExp.prototype.test (*S*)
 - 22.2.5.16 RegExp.prototype.toString ()
 - 22.2.5.17 get RegExp.prototype.unicode
- 22.2.6 Properties of RegExp Instances
 - 22.2.6.1 lastIndex
- 22.2.7 RegExp String Iterator Objects
 - 22.2.7.1 CreateRegExpStringIterator (*R*, *S*, *global*, *fullUnicode*)
 - 22.2.7.2 The %RegExpStringIteratorPrototype% Object
 - 22.2.7.2.1 %RegExpStringIteratorPrototype%.next ()
 - 22.2.7.2.2 %RegExpStringIteratorPrototype% [@@toStringTag]

23 Indexed Collections

- 23.1 Array Objects
 - 23.1.1 The Array Constructor
 - 23.1.1.1 Array (...*values*)
 - 23.1.2 Properties of the Array Constructor
 - 23.1.2.1 Array.from (*items* [, *mapfn* [, *thisArg*]])
 - 23.1.2.2 Array.isArray (*arg*)
 - 23.1.2.3 Array.of (...*items*)
 - 23.1.2.4 Array.prototype
 - 23.1.2.5 get Array [@@species]
 - 23.1.3 Properties of the Array Prototype Object
 - 23.1.3.1 Array.prototype.concat (...*items*)
 - 23.1.3.1.1 IsConcatSpreadable (*O*)
 - 23.1.3.2 Array.prototype.constructor
 - 23.1.3.3 Array.prototype.copyWithin (*target*, *start* [, *end*])
 - 23.1.3.4 Array.prototype.entries ()
 - 23.1.3.5 Array.prototype.every (*callbackfn* [, *thisArg*])
 - 23.1.3.6 Array.prototype.fill (*value* [, *start* [, *end*]])
 - 23.1.3.7 Array.prototype.filter (*callbackfn* [, *thisArg*])

- 23.1.3.8 `Array.prototype.find` (*predicate* [, *thisArg*])
 - 23.1.3.9 `Array.prototype.findIndex` (*predicate* [, *thisArg*])
 - 23.1.3.10 `Array.prototype.flat` ([*depth*])
 - 23.1.3.10.1 `FlattenIntoArray` (*target*, *source*, *sourceLen*, *start*, *depth* [, *mapperFunction*, *thisArg*])
 - 23.1.3.11 `Array.prototype.flatMap` (*mapperFunction* [, *thisArg*])
 - 23.1.3.12 `Array.prototype.forEach` (*callbackfn* [, *thisArg*])
 - 23.1.3.13 `Array.prototype.includes` (*searchElement* [, *fromIndex*])
 - 23.1.3.14 `Array.prototype.indexOf` (*searchElement* [, *fromIndex*])
 - 23.1.3.15 `Array.prototype.join` (*separator*)
 - 23.1.3.16 `Array.prototype.keys` ()
 - 23.1.3.17 `Array.prototype.lastIndexOf` (*searchElement* [, *fromIndex*])
 - 23.1.3.18 `Array.prototype.map` (*callbackfn* [, *thisArg*])
 - 23.1.3.19 `Array.prototype.pop` ()
 - 23.1.3.20 `Array.prototype.push` (...*items*)
 - 23.1.3.21 `Array.prototype.reduce` (*callbackfn* [, *initialValue*])
 - 23.1.3.22 `Array.prototype.reduceRight` (*callbackfn* [, *initialValue*])
 - 23.1.3.23 `Array.prototype.reverse` ()
 - 23.1.3.24 `Array.prototype.shift` ()
 - 23.1.3.25 `Array.prototype.slice` (*start*, *end*)
 - 23.1.3.26 `Array.prototype.some` (*callbackfn* [, *thisArg*])
 - 23.1.3.27 `Array.prototype.sort` (*comparefn*)
 - 23.1.3.27.1 `SortCompare` (*x*, *y*)
 - 23.1.3.28 `Array.prototype.splice` (*start*, *deleteCount*, ...*items*)
 - 23.1.3.29 `Array.prototype.toLocaleString` ([*reserved1* [, *reserved2*]])
 - 23.1.3.30 `Array.prototype.toString` ()
 - 23.1.3.31 `Array.prototype.unshift` (...*items*)
 - 23.1.3.32 `Array.prototype.values` ()
 - 23.1.3.33 `Array.prototype [@@iterator]` ()
 - 23.1.3.34 `Array.prototype [@@unscopables]`
- 23.1.4 Properties of Array Instances
- 23.1.4.1 `length`
- 23.1.5 Array Iterator Objects
- 23.1.5.1 `CreateArrayIterator` (*array*, *kind*)
 - 23.1.5.2 The `%ArrayIteratorPrototype%` Object
 - 23.1.5.2.1 `%ArrayIteratorPrototype%.next` ()
 - 23.1.5.2.2 `%ArrayIteratorPrototype%` [`@@toStringTag`]
- 23.2 TypedArray Objects
- 23.2.1 The `%TypedArray%` Intrinsic Object
 - 23.2.1.1 `%TypedArray%` ()
 - 23.2.2 Properties of the `%TypedArray%` Intrinsic Object
 - 23.2.2.1 `%TypedArray%.from` (*source* [, *mapfn* [, *thisArg*]])
 - 23.2.2.2 `%TypedArray%.of` (...*items*)
 - 23.2.2.3 `%TypedArray%.prototype`
 - 23.2.2.4 `get %TypedArray%` [`@@species`]
 - 23.2.3 Properties of the `%TypedArray%` Prototype Object
 - 23.2.3.1 `get %TypedArray%.prototype.buffer`
 - 23.2.3.2 `get %TypedArray%.prototype.byteLength`
 - 23.2.3.3 `get %TypedArray%.prototype.byteOffset`

- 23.2.3.4 %TypedArray%.prototype.constructor
- 23.2.3.5 %TypedArray%.prototype.copyWithin (*target*, *start* [, *end*])
- 23.2.3.6 %TypedArray%.prototype.entries ()
- 23.2.3.7 %TypedArray%.prototype.every (*callbackfn* [, *thisArg*])
- 23.2.3.8 %TypedArray%.prototype.fill (*value* [, *start* [, *end*]])
- 23.2.3.9 %TypedArray%.prototype.filter (*callbackfn* [, *thisArg*])
- 23.2.3.10 %TypedArray%.prototype.find (*predicate* [, *thisArg*])
- 23.2.3.11 %TypedArray%.prototype.findIndex (*predicate* [, *thisArg*])
- 23.2.3.12 %TypedArray%.prototype.forEach (*callbackfn* [, *thisArg*])
- 23.2.3.13 %TypedArray%.prototype.includes (*searchElement* [, *fromIndex*])
- 23.2.3.14 %TypedArray%.prototype.indexOf (*searchElement* [, *fromIndex*])
- 23.2.3.15 %TypedArray%.prototype.join (*separator*)
- 23.2.3.16 %TypedArray%.prototype.keys ()
- 23.2.3.17 %TypedArray%.prototype.lastIndexOf (*searchElement* [, *fromIndex*])
- 23.2.3.18 get %TypedArray%.prototype.length
- 23.2.3.19 %TypedArray%.prototype.map (*callbackfn* [, *thisArg*])
- 23.2.3.20 %TypedArray%.prototype.reduce (*callbackfn* [, *initialValue*])
- 23.2.3.21 %TypedArray%.prototype.reduceRight (*callbackfn* [, *initialValue*])
- 23.2.3.22 %TypedArray%.prototype.reverse ()
- 23.2.3.23 %TypedArray%.prototype.set (*source* [, *offset*])
 - 23.2.3.23.1 SetTypedArrayFromTypedArray (*target*, *targetOffset*, *source*)
 - 23.2.3.23.2 SetTypedArrayFromArrayLike (*target*, *targetOffset*, *source*)
- 23.2.3.24 %TypedArray%.prototype.slice (*start*, *end*)
- 23.2.3.25 %TypedArray%.prototype.some (*callbackfn* [, *thisArg*])
- 23.2.3.26 %TypedArray%.prototype.sort (*comparefn*)
- 23.2.3.27 %TypedArray%.prototype.subarray (*begin*, *end*)
- 23.2.3.28 %TypedArray%.prototype.toLocaleString ([*reserved1* [, *reserved2*]])
- 23.2.3.29 %TypedArray%.prototype.toString ()
- 23.2.3.30 %TypedArray%.prototype.values ()
- 23.2.3.31 %TypedArray%.prototype [@@iterator] ()
- 23.2.3.32 get %TypedArray%.prototype [@@toStringTag]
- 23.2.4 Abstract Operations for TypedArray Objects
 - 23.2.4.1 TypedArraySpeciesCreate (*exemplar*, *argumentList*)
 - 23.2.4.2 TypedArrayCreate (*constructor*, *argumentList*)
 - 23.2.4.3 ValidateTypedArray (*O*)
- 23.2.5 The *TypedArray* Constructors
 - 23.2.5.1 *TypedArray* (...*args*)
 - 23.2.5.1.1 AllocateTypedArray (*constructorName*, *newTarget*, *defaultProto* [, *length*])
 - 23.2.5.1.2 InitializeTypedArrayFromTypedArray (*O*, *srcArray*)
 - 23.2.5.1.3 InitializeTypedArrayFromArrayBuffer (*O*, *buffer*, *byteOffset*, *length*)
 - 23.2.5.1.4 InitializeTypedArrayFromList (*O*, *values*)
 - 23.2.5.1.5 InitializeTypedArrayFromArrayLike (*O*, *arrayLike*)
 - 23.2.5.1.6 AllocateTypedArrayBuffer (*O*, *length*)
- 23.2.6 Properties of the *TypedArray* Constructors
 - 23.2.6.1 *TypedArray*.BYTES_PER_ELEMENT
 - 23.2.6.2 *TypedArray*.prototype
- 23.2.7 Properties of the *TypedArray* Prototype Objects
 - 23.2.7.1 *TypedArray*.prototype.BYTES_PER_ELEMENT

23.2.7.2 *TypedArray*.prototype.constructor

23.2.8 Properties of *TypedArray* Instances

24 Keyed Collections

24.1 Map Objects

24.1.1 The Map Constructor

24.1.1.1 Map ([*iterable*])

24.1.1.2 AddEntriesFromIterable (*target*, *iterable*, *adder*)

24.1.2 Properties of the Map Constructor

24.1.2.1 Map.prototype

24.1.2.2 get Map [@@species]

24.1.3 Properties of the Map Prototype Object

24.1.3.1 Map.prototype.clear ()

24.1.3.2 Map.prototype.constructor

24.1.3.3 Map.prototype.delete (*key*)

24.1.3.4 Map.prototype.entries ()

24.1.3.5 Map.prototype.forEach (*callbackfn* [, *thisArg*])

24.1.3.6 Map.prototype.get (*key*)

24.1.3.7 Map.prototype.has (*key*)

24.1.3.8 Map.prototype.keys ()

24.1.3.9 Map.prototype.set (*key*, *value*)

24.1.3.10 get Map.prototype.size

24.1.3.11 Map.prototype.values ()

24.1.3.12 Map.prototype [@@iterator] ()

24.1.3.13 Map.prototype [@@toStringTag]

24.1.4 Properties of Map Instances

24.1.5 Map Iterator Objects

24.1.5.1 CreateMapIterator (*map*, *kind*)

24.1.5.2 The %MapIteratorPrototype% Object

24.1.5.2.1 %MapIteratorPrototype%.next ()

24.1.5.2.2 %MapIteratorPrototype% [@@toStringTag]

24.2 Set Objects

24.2.1 The Set Constructor

24.2.1.1 Set ([*iterable*])

24.2.2 Properties of the Set Constructor

24.2.2.1 Set.prototype

24.2.2.2 get Set [@@species]

24.2.3 Properties of the Set Prototype Object

24.2.3.1 Set.prototype.add (*value*)

24.2.3.2 Set.prototype.clear ()

24.2.3.3 Set.prototype.constructor

24.2.3.4 Set.prototype.delete (*value*)

24.2.3.5 Set.prototype.entries ()

24.2.3.6 Set.prototype.forEach (*callbackfn* [, *thisArg*])

24.2.3.7 Set.prototype.has (*value*)

24.2.3.8 Set.prototype.keys ()

24.2.3.9 get Set.prototype.size

24.2.3.10 Set.prototype.values ()

24.2.3.11 Set.prototype [@@iterator] ()

- 24.2.3.12 Set.prototype [@@toStringTag]
- 24.2.4 Properties of Set Instances
- 24.2.5 Set Iterator Objects
 - 24.2.5.1 CreateSetIterator (*set*, *kind*)
 - 24.2.5.2 The %SetIteratorPrototype% Object
 - 24.2.5.2.1 %SetIteratorPrototype%.next ()
 - 24.2.5.2.2 %SetIteratorPrototype% [@@toStringTag]
- 24.3 WeakMap Objects
 - 24.3.1 The WeakMap Constructor
 - 24.3.1.1 WeakMap ([*iterable*])
 - 24.3.2 Properties of the WeakMap Constructor
 - 24.3.2.1 WeakMap.prototype
 - 24.3.3 Properties of the WeakMap Prototype Object
 - 24.3.3.1 WeakMap.prototype.constructor
 - 24.3.3.2 WeakMap.prototype.delete (*key*)
 - 24.3.3.3 WeakMap.prototype.get (*key*)
 - 24.3.3.4 WeakMap.prototype.has (*key*)
 - 24.3.3.5 WeakMap.prototype.set (*key*, *value*)
 - 24.3.3.6 WeakMap.prototype [@@toStringTag]
 - 24.3.4 Properties of WeakMap Instances
- 24.4 WeakSet Objects
 - 24.4.1 The WeakSet Constructor
 - 24.4.1.1 WeakSet ([*iterable*])
 - 24.4.2 Properties of the WeakSet Constructor
 - 24.4.2.1 WeakSet.prototype
 - 24.4.3 Properties of the WeakSet Prototype Object
 - 24.4.3.1 WeakSet.prototype.add (*value*)
 - 24.4.3.2 WeakSet.prototype.constructor
 - 24.4.3.3 WeakSet.prototype.delete (*value*)
 - 24.4.3.4 WeakSet.prototype.has (*value*)
 - 24.4.3.5 WeakSet.prototype [@@toStringTag]
 - 24.4.4 Properties of WeakSet Instances
- 25 Structured Data
 - 25.1 ArrayBuffer Objects
 - 25.1.1 Notation
 - 25.1.2 Abstract Operations For ArrayBuffer Objects
 - 25.1.2.1 AllocateArrayBuffer (*constructor*, *byteLength*)
 - 25.1.2.2 IsDetachedBuffer (*arrayBuffer*)
 - 25.1.2.3 DetachArrayBuffer (*arrayBuffer* [, *key*])
 - 25.1.2.4 CloneArrayBuffer (*srcBuffer*, *srcByteOffset*, *srcLength*, *cloneConstructor*)
 - 25.1.2.5 IsUnsignedElementType (*type*)
 - 25.1.2.6 IsUnclampedIntegerElementType (*type*)
 - 25.1.2.7 IsBigIntElementType (*type*)
 - 25.1.2.8 IsNoTearConfiguration (*type*, *order*)
 - 25.1.2.9 RawBytesToNumeric (*type*, *rawBytes*, *isLittleEndian*)
 - 25.1.2.10 GetValueFromBuffer (*arrayBuffer*, *byteIndex*, *type*, *isTypedArray*, *order* [, *isLittleEndian*])
 - 25.1.2.11 NumericToRawBytes (*type*, *value*, *isLittleEndian*)
 - 25.1.2.12 SetValueInBuffer (*arrayBuffer*, *byteIndex*, *type*, *value*, *isTypedArray*, *order* [, *isLittleEndian*])

- 25.1.2.13 GetModifySetValueInBuffer (*arrayBuffer*, *byteIndex*, *type*, *value*, *op* [, *isLittleEndian*])
- 25.1.3 The ArrayBuffer Constructor
 - 25.1.3.1 ArrayBuffer (*length*)
- 25.1.4 Properties of the ArrayBuffer Constructor
 - 25.1.4.1 ArrayBuffer.isView (*arg*)
 - 25.1.4.2 ArrayBuffer.prototype
 - 25.1.4.3 get ArrayBuffer [@@species]
- 25.1.5 Properties of the ArrayBuffer Prototype Object
 - 25.1.5.1 get ArrayBuffer.prototype.byteLength
 - 25.1.5.2 ArrayBuffer.prototype.constructor
 - 25.1.5.3 ArrayBuffer.prototype.slice (*start*, *end*)
 - 25.1.5.4 ArrayBuffer.prototype [@@toStringTag]
- 25.1.6 Properties of ArrayBuffer Instances
- 25.2 SharedArrayBuffer Objects
 - 25.2.1 Abstract Operations for SharedArrayBuffer Objects
 - 25.2.1.1 AllocateSharedArrayBuffer (*constructor*, *byteLength*)
 - 25.2.1.2 IsSharedArrayBuffer (*obj*)
 - 25.2.2 The SharedArrayBuffer Constructor
 - 25.2.2.1 SharedArrayBuffer ([*length*])
 - 25.2.3 Properties of the SharedArrayBuffer Constructor
 - 25.2.3.1 SharedArrayBuffer.prototype
 - 25.2.3.2 get SharedArrayBuffer [@@species]
 - 25.2.4 Properties of the SharedArrayBuffer Prototype Object
 - 25.2.4.1 get SharedArrayBuffer.prototype.byteLength
 - 25.2.4.2 SharedArrayBuffer.prototype.constructor
 - 25.2.4.3 SharedArrayBuffer.prototype.slice (*start*, *end*)
 - 25.2.4.4 SharedArrayBuffer.prototype [@@toStringTag]
 - 25.2.5 Properties of SharedArrayBuffer Instances
- 25.3 DataView Objects
 - 25.3.1 Abstract Operations For DataView Objects
 - 25.3.1.1 GetViewValue (*view*, *requestIndex*, *isLittleEndian*, *type*)
 - 25.3.1.2 SetViewValue (*view*, *requestIndex*, *isLittleEndian*, *type*, *value*)
 - 25.3.2 The DataView Constructor
 - 25.3.2.1 DataView (*buffer* [, *byteOffset* [, *byteLength*]])
 - 25.3.3 Properties of the DataView Constructor
 - 25.3.3.1 DataView.prototype
 - 25.3.4 Properties of the DataView Prototype Object
 - 25.3.4.1 get DataView.prototype.buffer
 - 25.3.4.2 get DataView.prototype.byteLength
 - 25.3.4.3 get DataView.prototype.byteOffset
 - 25.3.4.4 DataView.prototype.constructor
 - 25.3.4.5 DataView.prototype.getBigInt64 (*byteOffset* [, *littleEndian*])
 - 25.3.4.6 DataView.prototype.getBigUint64 (*byteOffset* [, *littleEndian*])
 - 25.3.4.7 DataView.prototype.getFloat32 (*byteOffset* [, *littleEndian*])
 - 25.3.4.8 DataView.prototype.getFloat64 (*byteOffset* [, *littleEndian*])
 - 25.3.4.9 DataView.prototype.getInt8 (*byteOffset*)
 - 25.3.4.10 DataView.prototype.getInt16 (*byteOffset* [, *littleEndian*])
 - 25.3.4.11 DataView.prototype.getInt32 (*byteOffset* [, *littleEndian*])

- 25.3.4.12 DataView.prototype.getUint8 (*byteOffset*)
- 25.3.4.13 DataView.prototype.getUint16 (*byteOffset* [, *littleEndian*])
- 25.3.4.14 DataView.prototype.getUint32 (*byteOffset* [, *littleEndian*])
- 25.3.4.15 DataView.prototype.setBigInt64 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.16 DataView.prototype.setBigUint64 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.17 DataView.prototype.setFloat32 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.18 DataView.prototype.setFloat64 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.19 DataView.prototype.setInt8 (*byteOffset*, *value*)
- 25.3.4.20 DataView.prototype.setInt16 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.21 DataView.prototype.setInt32 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.22 DataView.prototype.setUint8 (*byteOffset*, *value*)
- 25.3.4.23 DataView.prototype.setUint16 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.24 DataView.prototype.setUint32 (*byteOffset*, *value* [, *littleEndian*])
- 25.3.4.25 DataView.prototype [@@toStringTag]
- 25.3.5 Properties of DataView Instances
- 25.4 The Atomics Object
 - 25.4.1 Abstract Operations for Atomics
 - 25.4.1.1 ValidateIntegerTypedArray (*typedArray* [, *waitable*])
 - 25.4.1.2 ValidateAtomicAccess (*typedArray*, *requestIndex*)
 - 25.4.1.3 GetWaiterList (*block*, *i*)
 - 25.4.1.4 EnterCriticalSection (*WL*)
 - 25.4.1.5 LeaveCriticalSection (*WL*)
 - 25.4.1.6 AddWaiter (*WL*, *W*)
 - 25.4.1.7 RemoveWaiter (*WL*, *W*)
 - 25.4.1.8 RemoveWaiters (*WL*, *c*)
 - 25.4.1.9 SuspendAgent (*WL*, *W*, *timeout*)
 - 25.4.1.10 NotifyWaiter (*WL*, *W*)
 - 25.4.1.11 AtomicReadModifyWrite (*typedArray*, *index*, *value*, *op*)
 - 25.4.1.12 ByteListBitwiseOp (*op*, *xBytes*, *yBytes*)
 - 25.4.1.13 ByteListEqual (*xBytes*, *yBytes*)
 - 25.4.2 Atomics.add (*typedArray*, *index*, *value*)
 - 25.4.3 Atomics.and (*typedArray*, *index*, *value*)
 - 25.4.4 Atomics.compareExchange (*typedArray*, *index*, *expectedValue*, *replacementValue*)
 - 25.4.5 Atomics.exchange (*typedArray*, *index*, *value*)
 - 25.4.6 Atomics.isLockFree (*size*)
 - 25.4.7 Atomics.load (*typedArray*, *index*)
 - 25.4.8 Atomics.or (*typedArray*, *index*, *value*)
 - 25.4.9 Atomics.store (*typedArray*, *index*, *value*)
 - 25.4.10 Atomics.sub (*typedArray*, *index*, *value*)
 - 25.4.11 Atomics.wait (*typedArray*, *index*, *value*, *timeout*)
 - 25.4.12 Atomics.notify (*typedArray*, *index*, *count*)
 - 25.4.13 Atomics.xor (*typedArray*, *index*, *value*)
 - 25.4.14 Atomics [@@toStringTag]
- 25.5 The JSON Object
 - 25.5.1 JSON.parse (*text* [, *reviver*])
 - 25.5.1.1 InternalizeJSONProperty (*holder*, *name*, *reviver*)
 - 25.5.2 JSON.stringify (*value* [, *replacer* [, *space*]])
 - 25.5.2.1 SerializeJSONProperty (*state*, *key*, *holder*)

- 25.5.2.2 QuoteJSONString (*value*)
- 25.5.2.3 UnicodeEscape (*C*)
- 25.5.2.4 SerializeJSONObject (*state, value*)
- 25.5.2.5 SerializeJSONArray (*state, value*)
- 25.5.3 JSON [@@toStringTag]
- 26 Managing Memory
 - 26.1 WeakRef Objects
 - 26.1.1 The WeakRef Constructor
 - 26.1.1.1 WeakRef (*target*)
 - 26.1.2 Properties of the WeakRef Constructor
 - 26.1.2.1 WeakRef.prototype
 - 26.1.3 Properties of the WeakRef Prototype Object
 - 26.1.3.1 WeakRef.prototype.constructor
 - 26.1.3.2 WeakRef.prototype.deref ()
 - 26.1.3.3 WeakRef.prototype [@@toStringTag]
 - 26.1.4 WeakRef Abstract Operations
 - 26.1.4.1 WeakRefDeref (*weakRef*)
 - 26.1.5 Properties of WeakRef Instances
 - 26.2 FinalizationRegistry Objects
 - 26.2.1 The FinalizationRegistry Constructor
 - 26.2.1.1 FinalizationRegistry (*cleanupCallback*)
 - 26.2.2 Properties of the FinalizationRegistry Constructor
 - 26.2.2.1 FinalizationRegistry.prototype
 - 26.2.3 Properties of the FinalizationRegistry Prototype Object
 - 26.2.3.1 FinalizationRegistry.prototype.constructor
 - 26.2.3.2 FinalizationRegistry.prototype.register (*target, heldValue* [, *unregisterToken*])
 - 26.2.3.3 FinalizationRegistry.prototype.unregister (*unregisterToken*)
 - 26.2.3.4 FinalizationRegistry.prototype [@@toStringTag]
 - 26.2.4 Properties of FinalizationRegistry Instances
- 27 Control Abstraction Objects
 - 27.1 Iteration
 - 27.1.1 Common Iteration Interfaces
 - 27.1.1.1 The *Iterable* Interface
 - 27.1.1.2 The *Iterator* Interface
 - 27.1.1.3 The *AsyncIterable* Interface
 - 27.1.1.4 The *AsyncIterator* Interface
 - 27.1.1.5 The *IteratorResult* Interface
 - 27.1.2 The %IteratorPrototype% Object
 - 27.1.2.1 %IteratorPrototype% [@@iterator] ()
 - 27.1.3 The %AsyncIteratorPrototype% Object
 - 27.1.3.1 %AsyncIteratorPrototype% [@@asyncIterator] ()
 - 27.1.4 Async-from-Sync Iterator Objects
 - 27.1.4.1 CreateAsyncFromSyncIterator (*syncIteratorRecord*)
 - 27.1.4.2 The %AsyncFromSyncIteratorPrototype% Object
 - 27.1.4.2.1 %AsyncFromSyncIteratorPrototype%.next ([*value*])
 - 27.1.4.2.2 %AsyncFromSyncIteratorPrototype%.return ([*value*])
 - 27.1.4.2.3 %AsyncFromSyncIteratorPrototype%.throw ([*value*])
 - 27.1.4.2.4 Async-from-Sync Iterator Value Unwrap Functions

- 27.1.4.3 Properties of Async-from-Sync Iterator Instances
- 27.1.4.4 AsyncFromSyncIteratorContinuation (*result, promiseCapability*)
- 27.2 Promise Objects
 - 27.2.1 Promise Abstract Operations
 - 27.2.1.1 PromiseCapability Records
 - 27.2.1.1.1 IfAbruptRejectPromise (*value, capability*)
 - 27.2.1.2 PromiseReaction Records
 - 27.2.1.3 CreateResolvingFunctions (*promise*)
 - 27.2.1.3.1 Promise Reject Functions
 - 27.2.1.3.2 Promise Resolve Functions
 - 27.2.1.4 FulfillPromise (*promise, value*)
 - 27.2.1.5 NewPromiseCapability (*C*)
 - 27.2.1.5.1 GetCapabilitiesExecutor Functions
 - 27.2.1.6 IsPromise (*x*)
 - 27.2.1.7 RejectPromise (*promise, reason*)
 - 27.2.1.8 TriggerPromiseReactions (*reactions, argument*)
 - 27.2.1.9 HostPromiseRejectionTracker (*promise, operation*)
 - 27.2.2 Promise Jobs
 - 27.2.2.1 NewPromiseReactionJob (*reaction, argument*)
 - 27.2.2.2 NewPromiseResolveThenableJob (*promiseToResolve, thenable, then*)
 - 27.2.3 The Promise Constructor
 - 27.2.3.1 Promise (*executor*)
 - 27.2.4 Properties of the Promise Constructor
 - 27.2.4.1 Promise.all (*iterable*)
 - 27.2.4.1.1 GetPromiseResolve (*promiseConstructor*)
 - 27.2.4.1.2 PerformPromiseAll (*iteratorRecord, constructor, resultCapability, promiseResolve*)
 - 27.2.4.1.3 **Promise.all** Resolve Element Functions
 - 27.2.4.2 Promise.allSettled (*iterable*)
 - 27.2.4.2.1 PerformPromiseAllSettled (*iteratorRecord, constructor, resultCapability, promiseResolve*)
 - 27.2.4.2.2 **Promise.allSettled** Resolve Element Functions
 - 27.2.4.2.3 **Promise.allSettled** Reject Element Functions
 - 27.2.4.3 Promise.any (*iterable*)
 - 27.2.4.3.1 PerformPromiseAny (*iteratorRecord, constructor, resultCapability, promiseResolve*)
 - 27.2.4.3.2 **Promise.any** Reject Element Functions
 - 27.2.4.4 Promise.prototype
 - 27.2.4.5 Promise.race (*iterable*)
 - 27.2.4.5.1 PerformPromiseRace (*iteratorRecord, constructor, resultCapability, promiseResolve*)
 - 27.2.4.6 Promise.reject (*r*)
 - 27.2.4.7 Promise.resolve (*x*)
 - 27.2.4.7.1 PromiseResolve (*C, x*)
 - 27.2.4.8 get Promise [@@species]
 - 27.2.5 Properties of the Promise Prototype Object
 - 27.2.5.1 Promise.prototype.catch (*onRejected*)
 - 27.2.5.2 Promise.prototype.constructor
 - 27.2.5.3 Promise.prototype.finally (*onFinally*)
 - 27.2.5.3.1 Then Finally Functions
 - 27.2.5.3.2 Catch Finally Functions
 - 27.2.5.4 Promise.prototype.then (*onFulfilled, onRejected*)

- 27.2.5.4.1 PerformPromiseThen (*promise*, *onFulfilled*, *onRejected* [, *resultCapability*])
- 27.2.5.5 Promise.prototype [@@toStringTag]
- 27.2.6 Properties of Promise Instances
- 27.3 GeneratorFunction Objects
 - 27.3.1 The GeneratorFunction Constructor
 - 27.3.1.1 GeneratorFunction (*p1*, *p2*, ... , *pn*, *body*)
 - 27.3.2 Properties of the GeneratorFunction Constructor
 - 27.3.2.1 GeneratorFunction.length
 - 27.3.2.2 GeneratorFunction.prototype
 - 27.3.3 Properties of the GeneratorFunction Prototype Object
 - 27.3.3.1 GeneratorFunction.prototype.constructor
 - 27.3.3.2 GeneratorFunction.prototype.prototype
 - 27.3.3.3 GeneratorFunction.prototype [@@toStringTag]
 - 27.3.4 GeneratorFunction Instances
 - 27.3.4.1 length
 - 27.3.4.2 name
 - 27.3.4.3 prototype
- 27.4 AsyncGeneratorFunction Objects
 - 27.4.1 The AsyncGeneratorFunction Constructor
 - 27.4.1.1 AsyncGeneratorFunction (*p1*, *p2*, ... , *pn*, *body*)
 - 27.4.2 Properties of the AsyncGeneratorFunction Constructor
 - 27.4.2.1 AsyncGeneratorFunction.length
 - 27.4.2.2 AsyncGeneratorFunction.prototype
 - 27.4.3 Properties of the AsyncGeneratorFunction Prototype Object
 - 27.4.3.1 AsyncGeneratorFunction.prototype.constructor
 - 27.4.3.2 AsyncGeneratorFunction.prototype.prototype
 - 27.4.3.3 AsyncGeneratorFunction.prototype [@@toStringTag]
 - 27.4.4 AsyncGeneratorFunction Instances
 - 27.4.4.1 length
 - 27.4.4.2 name
 - 27.4.4.3 prototype
- 27.5 Generator Objects
 - 27.5.1 Properties of the Generator Prototype Object
 - 27.5.1.1 Generator.prototype.constructor
 - 27.5.1.2 Generator.prototype.next (*value*)
 - 27.5.1.3 Generator.prototype.return (*value*)
 - 27.5.1.4 Generator.prototype.throw (*exception*)
 - 27.5.1.5 Generator.prototype [@@toStringTag]
 - 27.5.2 Properties of Generator Instances
 - 27.5.3 Generator Abstract Operations
 - 27.5.3.1 GeneratorStart (*generator*, *generatorBody*)
 - 27.5.3.2 GeneratorValidate (*generator*, *generatorBrand*)
 - 27.5.3.3 GeneratorResume (*generator*, *value*, *generatorBrand*)
 - 27.5.3.4 GeneratorResumeAbrupt (*generator*, *abruptCompletion*, *generatorBrand*)
 - 27.5.3.5 GetGeneratorKind ()
 - 27.5.3.6 GeneratorYield (*iterNextObj*)
 - 27.5.3.7 Yield (*value*)
 - 27.5.3.8 CreateIteratorFromClosure (*closure*, *generatorBrand*, *generatorPrototype*)

27.6 AsyncGenerator Objects

27.6.1 Properties of the AsyncGenerator Prototype Object

27.6.1.1 AsyncGenerator.prototype.constructor

27.6.1.2 AsyncGenerator.prototype.next (*value*)

27.6.1.3 AsyncGenerator.prototype.return (*value*)

27.6.1.4 AsyncGenerator.prototype.throw (*exception*)

27.6.1.5 AsyncGenerator.prototype [@@toStringTag]

27.6.2 Properties of AsyncGenerator Instances

27.6.3 AsyncGenerator Abstract Operations

27.6.3.1 AsyncGeneratorRequest Records

27.6.3.2 AsyncGeneratorStart (*generator*, *generatorBody*)

27.6.3.3 AsyncGeneratorValidate (*generator*, *generatorBrand*)

27.6.3.4 AsyncGeneratorResolve (*generator*, *value*, *done*)

27.6.3.5 AsyncGeneratorReject (*generator*, *exception*)

27.6.3.6 AsyncGeneratorResumeNext (*generator*)

27.6.3.6.1 AsyncGeneratorResumeNext Return Processor Fulfilled Functions

27.6.3.6.2 AsyncGeneratorResumeNext Return Processor Rejected Functions

27.6.3.7 AsyncGeneratorEnqueue (*generator*, *completion*, *generatorBrand*)

27.6.3.8 AsyncGeneratorYield (*value*)

27.6.3.9 CreateAsyncIteratorFromClosure (*closure*, *generatorBrand*, *generatorPrototype*)

27.7 AsyncFunction Objects

27.7.1 The AsyncFunction Constructor

27.7.1.1 AsyncFunction (*p1*, *p2*, ... , *pn*, *body*)

27.7.2 Properties of the AsyncFunction Constructor

27.7.2.1 AsyncFunction.length

27.7.2.2 AsyncFunction.prototype

27.7.3 Properties of the AsyncFunction Prototype Object

27.7.3.1 AsyncFunction.prototype.constructor

27.7.3.2 AsyncFunction.prototype [@@toStringTag]

27.7.4 AsyncFunction Instances

27.7.4.1 length

27.7.4.2 name

27.7.5 Async Functions Abstract Operations

27.7.5.1 AsyncFunctionStart (*promiseCapability*, *asyncFunctionBody*)

28 Reflection

28.1 The Reflect Object

28.1.1 Reflect.apply (*target*, *thisArgument*, *argumentsList*)

28.1.2 Reflect.construct (*target*, *argumentsList* [, *newTarget*])

28.1.3 Reflect.defineProperty (*target*, *propertyKey*, *attributes*)

28.1.4 Reflect.deleteProperty (*target*, *propertyKey*)

28.1.5 Reflect.get (*target*, *propertyKey* [, *receiver*])

28.1.6 Reflect.getOwnPropertyDescriptor (*target*, *propertyKey*)

28.1.7 Reflect.getPrototypeOf (*target*)

28.1.8 Reflect.has (*target*, *propertyKey*)

28.1.9 Reflect.isExtensible (*target*)

28.1.10 Reflect.ownKeys (*target*)

28.1.11 Reflect.preventExtensions (*target*)

28.1.12 Reflect.set (*target*, *propertyKey*, *V* [, *receiver*])

- 28.1.13 Reflect.setPrototypeOf (*target, proto*)
- 28.1.14 Reflect [@@toStringTag]
- 28.2 Proxy Objects
 - 28.2.1 The Proxy Constructor
 - 28.2.1.1 Proxy (*target, handler*)
 - 28.2.2 Properties of the Proxy Constructor
 - 28.2.2.1 Proxy.revocable (*target, handler*)
 - 28.2.2.1.1 Proxy Revocation Functions
- 28.3 Module Namespace Objects
 - 28.3.1 @@toStringTag
- 29 Memory Model
 - 29.1 Memory Model Fundamentals
 - 29.2 Agent Events Records
 - 29.3 Chosen Value Records
 - 29.4 Candidate Executions
 - 29.5 Abstract Operations for the Memory Model
 - 29.5.1 EventSet (*execution*)
 - 29.5.2 SharedDataBlockEventSet (*execution*)
 - 29.5.3 HostEventSet (*execution*)
 - 29.5.4 ComposeWriteEventBytes (*execution, byteIndex, Ws*)
 - 29.5.5 ValueOfReadEvent (*execution, R*)
 - 29.6 Relations of Candidate Executions
 - 29.6.1 agent-order
 - 29.6.2 reads-bytes-from
 - 29.6.3 reads-from
 - 29.6.4 host-synchronizes-with
 - 29.6.5 synchronizes-with
 - 29.6.6 happens-before
 - 29.7 Properties of Valid Executions
 - 29.7.1 Valid Chosen Reads
 - 29.7.2 Coherent Reads
 - 29.7.3 Tear Free Reads
 - 29.7.4 Sequentially Consistent Atomics
 - 29.7.5 Valid Executions
 - 29.8 Races
 - 29.9 Data Races
 - 29.10 Data Race Freedom
 - 29.11 Shared Memory Guidelines
- A Grammar Summary
 - A.1 Lexical Grammar
 - A.2 Expressions
 - A.3 Statements
 - A.4 Functions and Classes
 - A.5 Scripts and Modules
 - A.6 Number Conversions
 - A.7 Universal Resource Identifier Character Classes
 - A.8 Regular Expressions
- B Additional ECMAScript Features for Web Browsers

B.1 Additional Syntax

B.1.1 Numeric Literals

B.1.1.1 Static Semantics

B.1.2 String Literals

B.1.2.1 Static Semantics

B.1.3 HTML-like Comments

B.1.4 Regular Expressions Patterns

B.1.4.1 SS: Early Errors

B.1.4.2 SS: IsCharacterClass

B.1.4.3 SS: CharacterValue

B.1.4.4 Pattern Semantics

B.1.4.4.1 CharacterRangeOrUnion (*A*, *B*)

B.2 Additional Built-in Properties

B.2.1 Additional Properties of the Global Object

B.2.1.1 escape (*string*)

B.2.1.2 unescape (*string*)

B.2.2 Additional Properties of the Object.prototype Object

B.2.2.1 Object.prototype.__proto__

B.2.2.1.1 get Object.prototype.__proto__

B.2.2.1.2 set Object.prototype.__proto__

B.2.2.2 Object.prototype.__defineGetter__ (*P*, *getter*)

B.2.2.3 Object.prototype.__defineSetter__ (*P*, *setter*)

B.2.2.4 Object.prototype.__lookupGetter__ (*P*)

B.2.2.5 Object.prototype.__lookupSetter__ (*P*)

B.2.3 Additional Properties of the String.prototype Object

B.2.3.1 String.prototype.substr (*start*, *length*)

B.2.3.2 String.prototype.anchor (*name*)

B.2.3.2.1 CreateHTML (*string*, *tag*, *attribute*, *value*)

B.2.3.3 String.prototype.big ()

B.2.3.4 String.prototype.blink ()

B.2.3.5 String.prototype.bold ()

B.2.3.6 String.prototype.fixed ()

B.2.3.7 String.prototype.fontcolor (*color*)

B.2.3.8 String.prototype.fontSize (*size*)

B.2.3.9 String.prototype.italics ()

B.2.3.10 String.prototype.link (*url*)

B.2.3.11 String.prototype.small ()

B.2.3.12 String.prototype.strike ()

B.2.3.13 String.prototype.sub ()

B.2.3.14 String.prototype.sup ()

B.2.3.15 String.prototype.trimLeft ()

B.2.3.16 String.prototype.trimRight ()

B.2.4 Additional Properties of the Date.prototype Object

B.2.4.1 Date.prototype.getYear ()

B.2.4.2 Date.prototype.setYear (*year*)

B.2.4.3 Date.prototype.toGMTString ()

B.2.5 Additional Properties of the RegExp.prototype Object

B.2.5.1 RegExp.prototype.compile (*pattern*, *flags*)

B.3 Other Additional Features

B.3.1 `__proto__` Property Names in Object Initializers

B.3.2 Labelled Function Declarations

B.3.3 Block-Level Function Declarations Web Legacy Compatibility Semantics

B.3.3.1 Changes to `FunctionDeclarationInstantiation`

B.3.3.2 Changes to `GlobalDeclarationInstantiation`

B.3.3.3 Changes to `EvalDeclarationInstantiation`

B.3.3.4 Changes to Block SS: Early Errors

B.3.3.5 Changes to **switch** Statement SS: Early Errors

B.3.3.6 Changes to `BlockDeclarationInstantiation`

B.3.4 `FunctionDeclarations` in `IfStatement` Statement Clauses

B.3.5 `VariableStatements` in Catch Blocks

B.3.6 Initializers in `ForIn` Statement Heads

B.3.7 The `[[IsHTMLDDA]]` Internal Slot

B.3.7.1 Changes to `ToBoolean`

B.3.7.2 Changes to Abstract Equality Comparison

B.3.7.3 Changes to the **typeof** Operator

C The Strict Mode of ECMAScript

D Host Layering Points

D.1 Host Hooks

D.2 Host-defined Fields

D.3 Host-defined Objects

D.4 Running Jobs

D.5 Internal Methods of Exotic Objects

D.6 Built-in Objects and Methods

E Corrections and Clarifications in ECMAScript 2015 with Possible Compatibility Impact

F Additions and Changes That Introduce Incompatibilities with Prior Editions

G Colophon

H Bibliography

I Copyright & Software License

Introduction

This Ecma Standard defines the ECMAScript 2021 Language. It is the twelfth edition of the ECMAScript Language Specification. Since publication of the first edition in 1997, ECMAScript has grown to be one of the world's most widely used general-purpose programming languages. It is best known as the language embedded in web browsers but has also been widely adopted for server and embedded applications.

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

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

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

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. However, that work was not completed and not published as the fourth edition of ECMAScript but some of it was incorporated into the development of the sixth edition.

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. The fifth edition was adopted by the Ecma General Assembly of December 2009.

The fifth edition was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262:2011. Edition 5.1 of the ECMAScript Standard incorporated minor corrections and is the same text as ISO/IEC 16262:2011. The 5.1 Edition was adopted by the Ecma General Assembly of June 2011.

Focused development of the sixth edition started in 2009, as the fifth edition was being prepared for publication. However, this was preceded by significant experimentation and language enhancement design efforts dating to the publication of the third edition in 1999. In a very real sense, the completion of the sixth edition is the culmination of a fifteen year effort. The goals for this edition included providing better support for large applications, library creation, and for use of ECMAScript as a compilation target for other languages. Some of its major enhancements included modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tail calls. The ECMAScript library of built-ins was expanded to support additional data abstractions including maps, sets, and arrays of binary numeric values as well as additional support for Unicode supplemental characters in strings and regular expressions. The built-ins were also made extensible via subclassing.

The sixth edition provides the foundation for regular, incremental language and library enhancements. The sixth edition was adopted by the General Assembly of June 2015.

ECMAScript 2016 was the first ECMAScript edition released under Ecma TC39's new yearly release cadence and open development process. A plain-text source document was built from the ECMAScript 2015 source document to serve as the base for further development entirely on GitHub. Over the year of this standard's development, hundreds of pull requests and issues were filed representing thousands of bug fixes, editorial fixes and other improvements. Additionally, numerous software tools were developed to aid in this effort including Ecm Markup, Ecm Markdown, and Grammarly. ES2016 also included support for a new exponentiation operator and adds a new method to **Array.prototype** called **includes**.

ECMAScript 2017 introduced Async Functions, Shared Memory, and Atomics along with smaller language and library enhancements, bug fixes, and editorial updates. Async functions improve the asynchronous programming experience by providing syntax for promise-returning functions. Shared Memory and Atomics introduce a new **memory model** that allows multi-agent programs to communicate using atomic operations that ensure a well-defined execution order even on parallel CPUs. It also included new static methods on Object: **Object.values**, **Object.entries**, and **Object.getOwnPropertyDescriptors**.

ECMAScript 2018 introduced support for asynchronous iteration via the AsyncIterator protocol and async generators. It also included four new regular expression features: the **dotAll** flag, named capture groups, Unicode property escapes, and look-behind assertions. Lastly it included object rest and spread properties.

ECMAScript 2019 introduced a few new built-in functions: **flat** and **flatMap** on **Array.prototype** for flattening arrays, **Object.fromEntries** for directly turning the return value of **Object.entries** into a new Object, and **trimStart** and **trimEnd** on **String.prototype** as better-named alternatives to the widely implemented but non-standard **String.prototype.trimLeft** and **trimRight** built-ins. In addition, it included a few minor updates to syntax and semantics. Updated syntax included optional catch binding parameters and allowing U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) in string literals to align with JSON. Other updates included requiring that **Array.prototype.sort** be a stable sort, requiring that **JSON.stringify** return well-formed UTF-8 regardless of input, and clarifying **Function.prototype.toString** by requiring that it either return the corresponding original source text or a standard placeholder.

ECMAScript 2020, the 11th edition, introduces the **matchAll** method for Strings, to produce an iterator for all match objects generated by a global regular expression; **import()**, a syntax to asynchronously import Modules with a dynamic specifier; **BigInt**, a new number primitive for working with arbitrary precision integers; **Promise.allSettled**, a new Promise combinator that does not short-circuit; **globalThis**, a universal way to access the global **this** value; dedicated **export * as ns from 'module'** syntax for use within modules; increased standardization of **for-in** enumeration order; **import.meta**, a **host**-populated object available in Modules that may contain contextual information about the Module; as well as adding two new syntax features to improve working with “nullish” values (**null** or **undefined**): nullish coalescing, a value selection operator; and optional chaining, a property access and function invocation operator that short-circuits if the value to access/invoke is nullish.

This specification, the 12th edition, introduces the **replaceAll** method for Strings; **Promise.any**, a Promise combinator that short-circuits when an input value is fulfilled; **AggregateError**, a new Error type to represent multiple errors at once; logical assignment operators (**??=**, **&&=**, **||=**); **WeakRef**, for referring to a target object without preserving it from garbage collection, and **FinalizationRegistry**, to manage registration and

unregistration of cleanup operations performed when target objects are garbage collected; separators for numeric literals (**1_000**); and **Array.prototype.sort** was made stable.

Dozens of individuals representing many organizations have made very significant contributions within Ecma TC39 to the development of this edition and to the prior editions. In addition, a vibrant community has emerged supporting TC39's ECMAScript efforts. This community has reviewed numerous drafts, filed thousands of bug reports, performed implementation experiments, contributed test suites, and educated the world-wide developer community about ECMAScript. Unfortunately, it is impossible to identify and acknowledge every person and organization who has contributed to this effort.

Allen Wirfs-Brock

ECMA-262, Project Editor, 6th Edition

Brian Terlson

ECMA-262, Project Editor, 7th through 10th Editions

Jordan Harband

ECMA-262, Project Editor, 10th through 12th Editions

1 Scope

This Standard defines the ECMAScript 2021 general-purpose programming 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 text input in conformance with the latest version of the Unicode Standard and ISO/IEC 10646.

A conforming implementation of ECMAScript that provides an application programming interface (API) 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 any “future reserved words” noted in subclause [12.6.2](#) of this specification.

A conforming implementation of ECMAScript must not implement any extension that is listed as a Forbidden Extension in subclause [17.1](#).

A conforming implementation of ECMAScript must not redefine any facilities that are not [implementation-defined](#), [implementation-approximated](#), or [host-defined](#).

A conforming implementation of ECMAScript may choose to implement or not implement *Normative Optional* subclauses. If any Normative Optional behaviour is implemented, all of the behaviour in the containing Normative Optional clause must be implemented. A Normative Optional clause is denoted in this specification with the words "Normative Optional" in a coloured box, as shown below.

NORMATIVE OPTIONAL

2.1 Example Clause Heading

Example clause contents.

3 Normative References

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

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

ECMA-402, *ECMAScript 2015 Internationalization API Specification*.
<https://ecma-international.org/publications/standards/Ecma-402.htm>

ECMA-404, *The JSON Data Interchange Format*.
<https://ecma-international.org/publications/standards/Ecma-404.htm>

4 Overview

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

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

ECMAScript was originally designed to be used as a scripting language, but has become widely used as a general-purpose programming language. 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.

ECMAScript was originally designed to be a *Web scripting language*, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript is now used 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](#).

ECMAScript usage has moved beyond simple scripting and it is now used for the full spectrum of programming tasks in many different environments and scales. As the usage of ECMAScript has expanded, so have the features and facilities it provides. ECMAScript is now a fully featured general-purpose programming language.

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 Hosts and Implementations

To aid integrating ECMAScript into [host environments](#), this specification defers the definition of certain facilities (e.g., [abstract operations](#)), either in whole or in part, to a source outside of this specification. Editorially, this specification distinguishes the following kinds of deferrals.

An *implementation* is an external source that further defines facilities enumerated in Annex D or those that are marked as [implementation-defined](#) or [implementation-approximated](#). In informal use, an implementation refers to a concrete artefact, such as a particular web browser.

An *implementation-defined* facility is one that defers its definition to an external source without further qualification. This specification does not make any recommendations for particular behaviours, and conforming implementations are free to choose any behaviour within the constraints put forth by this specification.

An *implementation-approximated* facility is one that defers its definition to an external source while recommending an ideal behaviour. While conforming implementations are free to choose any behaviour within the constraints put forth by this specification, they are encouraged to strive to approximate the ideal. Some mathematical operations, such as

Math.exp, are [implementation-approximated](#).

A *host* is an external source that further defines facilities listed in Annex D but does not further define other [implementation-defined](#) or [implementation-approximated](#) facilities. In informal use, a *host* refers to the set of all implementations, such as the set of all web browsers, that interface with this specification in the same way via Annex D. A *host* is often an external specification, such as WHATWG HTML (<https://html.spec.whatwg.org/>). In other words, facilities that are [host-defined](#) are often further defined in external specifications.

A *host hook* is an abstract operation that is defined in whole or in part by an external source. All *host* hooks must be listed in Annex D.

A *host-defined* facility is one that defers its definition to an external source without further qualification and is listed in Annex D. Implementations that are not hosts may also provide definitions for [host-defined](#) facilities.

A *host environment* is a particular choice of definition for all [host-defined](#) facilities. A *host environment* typically includes objects or functions which allow obtaining input and providing output as [host-defined](#) properties of the [global object](#).

This specification follows the editorial convention of always using the most specific term. For example, if a facility is [host-defined](#), it should not be referred to as [implementation-defined](#).

Both hosts and implementations may interface with this specification via the language types, specification types, [abstract operations](#), grammar productions, intrinsic objects, and intrinsic symbols defined herein.

4.3 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 *zero or more properties* each with *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 assign a different value to 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**, **BigInt**, **String**, and **Symbol**; an object is a member of the built-in type **Object**; and a function is a callable object. A function that is associated with an object via a property is called 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](#); objects that are fundamental to the [runtime semantics](#) of the language including **Object**, **Function**, **Boolean**, **Symbol**, and various **Error** objects; objects that represent and manipulate numeric values including **Math**, **Number**, and **Date**; the text processing objects **String** and **RegExp**; objects that are indexed collections of values including **Array** and nine different kinds of Typed Arrays whose elements all have a specific numeric data representation; keyed collections including **Map** and **Set** objects; objects supporting structured data including the **JSON** object, **ArrayBuffer**, **SharedArrayBuffer**, and **DataView**; objects supporting control abstractions including generator functions and **Promise** objects; and reflection objects including **Proxy** and **Reflect**.

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

bitwise operators, binary logical operators, assignment operators, and the comma operator.

Large ECMAScript programs are supported by *modules* which allow a program to be divided into multiple sequences of statements and declarations. Each module explicitly identifies declarations it uses that need to be provided by other modules and which of its declarations are available for use by other modules.

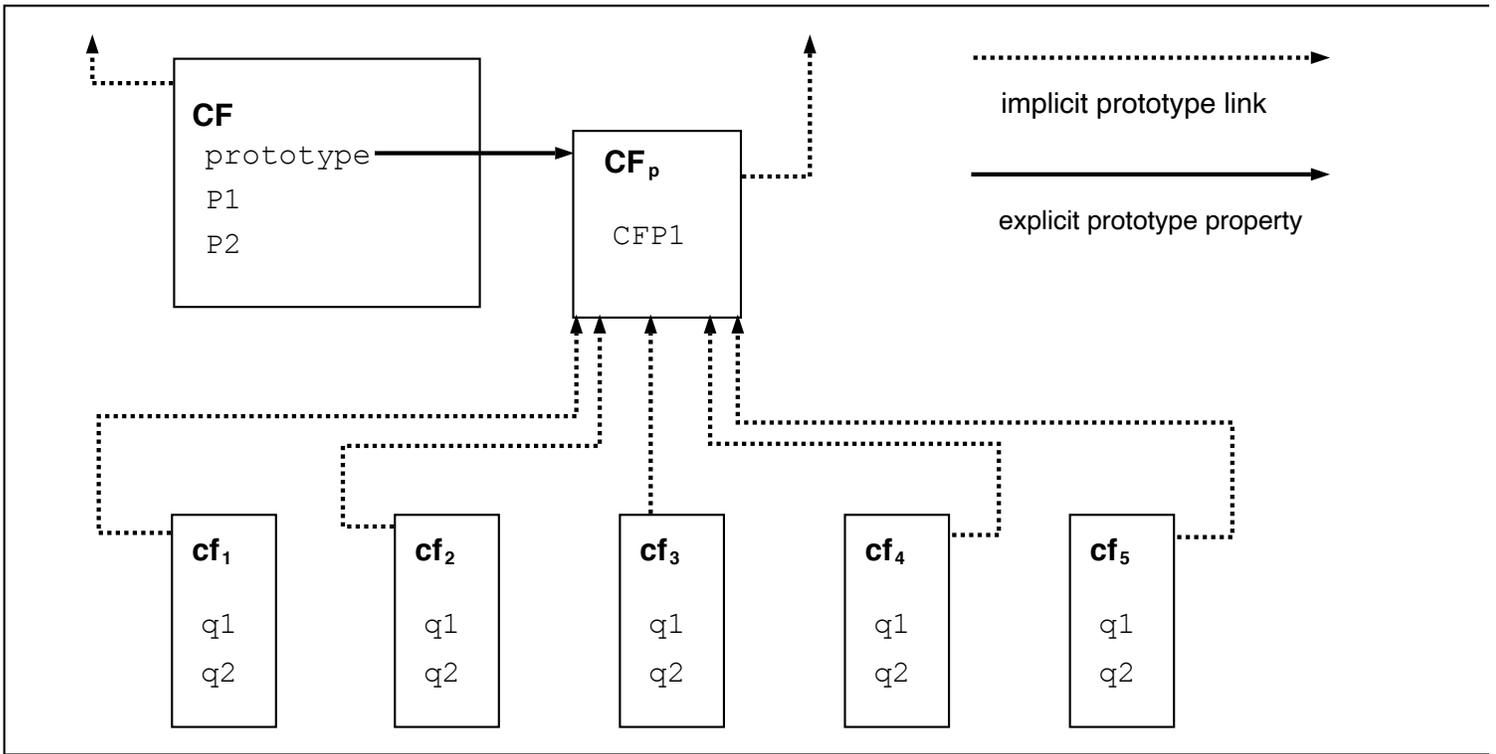
ECMAScript syntax intentionally resembles Java syntax. ECMAScript syntax is relaxed to enable it to serve as an easy-to-use scripting language. For example, a variable is not required to have its type declared nor are types associated with properties, and defined functions are not required to have their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

4.3.1 Objects

Even though ECMAScript includes syntax for class definitions, 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 *constructors* 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



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 **CF_p**. The **constructor**, **CF**, has two properties itself, named "P1" and "P2", which are not visible to **CF_p**, **cf₁**, **cf₂**, **cf₃**, **cf₄**, or **cf₅**. The property named "CFP1" in **CF_p** is shared by **cf₁**, **cf₂**, **cf₃**, **cf₄**, and **cf₅** (but not by **CF**), as are any properties found in **CF_p**'s implicit prototype chain that are not named "q1", "q2", or "CFP1". Notice that there is no implicit prototype link between **CF** and **CF_p**.

Unlike 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 **CF_p**.

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 objects themselves follow such a class-like pattern. Beginning with ECMAScript 2015, 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.

4.3.2 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 source text units as described in 11.2.2. Because strict mode is selected at the level of a syntactic source text unit, strict mode only imposes restrictions that have local effect within such a source text unit. Strict mode does not restrict or modify any aspect of the ECMAScript semantics that must operate consistently across multiple source text units. A complete ECMAScript program may be composed of both strict mode and non-strict mode ECMAScript source text units. In this case, strict mode only applies when actually executing code that is defined within a strict mode source text unit.

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

4.4 Terms and Definitions

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

4.4.1 implementation-approximated

an [implementation-approximated](#) facility is defined in whole or in part by an external source but has a recommended, ideal behaviour in this specification

4.4.2 implementation-defined

an [implementation-defined](#) facility is defined in whole or in part by an external source to this specification

4.4.3 host-defined

same as [implementation-defined](#)

NOTE Editorially, see clause 4.2.

4.4.4 type

set of data values as defined in clause 6

4.4.5 primitive value

member of one of the types Undefined, Null, Boolean, Number, BigInt, 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.

4.4.6 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.

4.4.7 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.

4.4.8 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.

4.4.9 ordinary object

object that has the default behaviour for the essential internal methods that must be supported by all objects

4.4.10 exotic object

object that does not have the default behaviour for one or more of the essential internal methods

NOTE Any object that is not an [ordinary object](#) is an [exotic object](#).

4.4.11 standard object

object whose semantics are defined by this specification

4.4.12 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](#).

4.4.13 undefined value

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

4.4.14 Undefined type

type whose sole value is the **undefined** value

4.4.15 null value

primitive value that represents the intentional absence of any object value

4.4.16 Null type

type whose sole value is the **null** value

4.4.17 Boolean value

member of the Boolean type

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

4.4.18 Boolean type

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

4.4.19 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.

4.4.20 String value

primitive value that is a finite ordered sequence of zero or more 16-bit unsigned [integer](#) values

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.

4.4.21 String type

set of all possible String values

4.4.22 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 (22.1.1.1).

4.4.23 Number value

primitive value corresponding to a double-precision 64-bit binary format [IEEE 754-2019](#) value

NOTE A [Number value](#) is a member of the Number type and is a direct representation of a number.

4.4.24 Number type

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

4.4.25 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 (21.1.1.1).

4.4.26 Infinity

[Number value](#) that is the positive infinite [Number value](#)

4.4.27 NaN

Number value that is an [IEEE 754-2019](#) “Not-a-Number” value

4.4.28 BigInt value

primitive value corresponding to an arbitrary-precision [integer](#) value

4.4.29 BigInt type

set of all possible BigInt values

4.4.30 BigInt object

member of the Object type that is an instance of the standard built-in BigInt [constructor](#)

4.4.31 Symbol value

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

4.4.32 Symbol type

set of all possible Symbol values

4.4.33 Symbol object

member of the Object type that is an instance of the standard built-in Symbol [constructor](#)

4.4.34 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.

4.4.35 built-in function

built-in object that is a function

NOTE Examples of built-in functions include **parseInt** and **Math.exp**. A [host](#) or implementation may provide additional built-in functions that are not described in this specification.

4.4.36 property

part of an object that associates a key (either a String value or a Symbol value) and a 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.

4.4.37 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.

4.4.38 built-in method

method that is a built-in function

NOTE Standard built-in methods are defined in this specification. A [host](#) or implementation may provide additional built-in methods that are not described in this specification.

4.4.39 attribute

internal value that defines some characteristic of a property

4.4.40 own property

property that is directly contained by its object

4.4.41 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

4.5 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](#) through [10](#) define the execution environment within which ECMAScript programs operate.

Clauses [11](#) through [17](#) define the actual ECMAScript programming language including its syntactic encoding and the execution semantics of all language features.

Clauses [18](#) through [28](#) define the ECMAScript standard library. They include the definitions of all of the standard objects that are available for use by ECMAScript programs as they execute.

Clause [29](#) describes the memory consistency model of accesses on SharedArrayBuffer-backed memory and methods

of the `Atomics` object.

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 12. This grammar has as its terminal symbols Unicode code points that conform to the rules for *SourceCharacter* defined in 11.1. It defines a set of productions, starting from the *goal symbol* *InputElementDiv*, *InputElementTemplateTail*, or *InputElementRegExp*, or *InputElementRegExpOrTemplateTail*, 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 (12.9). 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 22.2.1. 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 symbol* *Pattern*, 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 7.1.4.1.

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 13 through 16. 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 two alternative goal symbols *Script* and *Module*, that describe how sequences of tokens form syntactically correct independent components of ECMAScript programs.

When a stream of code points is to be parsed as an ECMAScript *Script* or *Module*, 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 input stream 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* or *Module*), with no tokens left over.

When a parse is successful, it constructs a *parse tree*, a rooted tree structure in which each node is a *Parse Node*. Each Parse Node is an *instance* of a symbol in the grammar; it represents a span of the source text that can be derived from that symbol. The root node of the parse tree, representing the whole of the source text, is an instance of the parse's *goal symbol*. When a Parse Node is an instance of a nonterminal, it is also an instance of some production that has that nonterminal as its left-hand side. Moreover, it has zero or more *children*, one for each symbol on the production's right-hand side: each child is a Parse Node that is an instance of the corresponding symbol.

New Parse Nodes are instantiated for each invocation of the parser and never reused between parses even of identical source text. Parse Nodes are considered *the same Parse Node* if and only if they represent the same span of source text, are instances of the same grammar symbol, and resulted from the same parser invocation.

NOTE 1 Parsing the same String multiple times will lead to different Parse Nodes. For example, consider:

```
let str = "1 + 1;";
eval(str);
eval(str);
```

Each call to **eval** converts the value of **str** into an ECMAScript source text and performs an independent parse that creates its own separate tree of Parse Nodes. The trees are distinct even though each parse operates upon a source text that was derived from the same String value.

NOTE 2 Parse Nodes are specification artefacts, and implementations are not required to use an analogous data structure.

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

The syntactic grammar as presented in clauses 13 through 16 is not a complete account of which token sequences are accepted as a correct ECMAScript *Script* or *Module*. 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 do not form a valid ECMAScript *Script* or *Module*. 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. Typically, an **early error** rule will then define an error condition if "*P* is not covering an *N*", where *P* is a Parse Node (an instance of the generalized production) and *N* is a nonterminal from the supplemental grammar. Here, the sequence of tokens originally matched by *P* is parsed again using *N* as the **goal symbol**. (If *N* takes grammatical parameters, then they are set to the same values used when *P* was originally parsed.) An error occurs if the sequence of tokens cannot be parsed as a single instance of *N*, with no tokens left over. Subsequently, algorithms access the result of the parse using a phrase of the form "the *N* that is covered by *P*". This will always be a Parse Node (an instance of *N*, unique for a given *P*), since any parsing failure would have been detected by an **early error** rule.

5.1.5 Grammar Notation

Terminal symbols 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. A code point in a terminal symbol cannot be expressed by a `\ UnicodeEscapeSequence`.

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 Initializeropt
```

is a convenient abbreviation for:

VariableDeclaration :
 BindingIdentifier
 BindingIdentifier *Initializer*

and that:

ForStatement :
 for (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

is a convenient abbreviation for:

ForStatement :
 for (*LexicalDeclaration* ; *Expression*_{opt}) *Statement*
 for (*LexicalDeclaration* *Expression* ; *Expression*_{opt}) *Statement*

which in turn is an abbreviation for:

ForStatement :
 for (*LexicalDeclaration* ;) *Statement*
 for (*LexicalDeclaration* ; *Expression*) *Statement*
 for (*LexicalDeclaration* *Expression* ;) *Statement*
 for (*LexicalDeclaration* *Expression* ; *Expression*) *Statement*

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

A production may be parameterized by a subscripted annotation of the form “[parameters]”, which may appear as a suffix to the nonterminal symbol defined by the production. “parameters” may be either a single name or a comma separated list of names. A 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:

*StatementList*_[Return] :
 ReturnStatement
 ExpressionStatement

is a convenient abbreviation for:

StatementList :
 ReturnStatement
 ExpressionStatement

StatementList_Return :
 ReturnStatement
 ExpressionStatement

and that:

*StatementList*_[Return, In] :
 ReturnStatement
 ExpressionStatement

is an abbreviation for:

```
StatementList :  
    ReturnStatement  
    ExpressionStatement
```

```
StatementList_Return :  
    ReturnStatement  
    ExpressionStatement
```

```
StatementList_In :  
    ReturnStatement  
    ExpressionStatement
```

```
StatementList_Return_In :  
    ReturnStatement  
    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:

```
StatementList :  
    ReturnStatement  
    ExpressionStatement [+In]
```

is equivalent to saying:

```
StatementList :  
    ReturnStatement  
    ExpressionStatement_In
```

and:

```
StatementList :  
    ReturnStatement  
    ExpressionStatement [~In]
```

is equivalent to:

```
StatementList :  
    ReturnStatement  
    ExpressionStatement
```

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

```
VariableDeclaration :  
    BindingIdentifier Initializer [+In] opt
```

is an abbreviation for:

VariableDeclaration :
 BindingIdentifier
 BindingIdentifier *Initializer_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:

*VariableDeclaration*_[In] :
 BindingIdentifier *Initializer*_[?In]

is an abbreviation for:

VariableDeclaration :
 BindingIdentifier *Initializer*

VariableDeclaration_In :
 BindingIdentifier *Initializer_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:

*StatementList*_[Return] :
 [+Return] *ReturnStatement*
 ExpressionStatement

is an abbreviation for:

StatementList :
 ExpressionStatement

StatementList_Return :
 ReturnStatement
 ExpressionStatement

and that:

*StatementList*_[Return] :
 [~Return] *ReturnStatement*
 ExpressionStatement

is an abbreviation for:

StatementList :
 ReturnStatement
 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 = *seq*]” appears in the right-hand side of a production, it indicates that the production may only be used if the token sequence *seq* is a prefix of the immediately following input token sequence. Similarly, “[lookahead ∈ *set*]”, where *set* is a finite nonempty set of token sequences, indicates that the production may only be used if some element of *set* is a prefix of the immediately following token sequence. For convenience, the set can also be written as a nonterminal, in which case it represents the set of all token sequences to which that nonterminal could expand. It is considered an editorial error if the nonterminal could expand to infinitely many distinct token sequences.

These conditions may be negated. “[lookahead ≠ *seq*]” indicates that the containing production may only be used if *seq* is *not* a prefix of the immediately following input token sequence, and “[lookahead ∉ *set*]” indicates that the production may only be used if *no* element of *set* is a prefix of the immediately following token sequence.

As an example, given the definitions:

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

```
DecimalDigits ::  
DecimalDigit  
DecimalDigits DecimalDigit
```

the definition:

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

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

followed by another decimal digit.

Note that when these phrases are used in the syntactic grammar, it may not be possible to unambiguously identify the immediately following token sequence because determining later tokens requires knowing which lexical *goal symbol* to use at later positions. As such, when these are used in the syntactic grammar, it is considered an editorial error for a token sequence *seq* to appear in a lookahead restriction (including as part of a set of sequences) if the choices of lexical goal symbols to use could change whether or not *seq* would be a prefix of the resulting token sequence.

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.

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 with an ordered, comma-separated sequence of alias names which may be used within the algorithm steps to reference the argument passed in that position. Optional parameters are denoted with surrounding brackets ([, *name*]) and are no different from required parameters within algorithm steps. A rest parameter may appear at the end of a parameter list, denoted with leading ellipsis (*...name*). The rest parameter

captures all of the arguments provided following the required and optional parameters into a [List](#). If there are no such additional arguments, that [List](#) is empty.

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 that begins with “*Assert:*” asserts 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.

Algorithm steps may declare named aliases for any value using the form “Let *x* be *someValue*”. These aliases are reference-like in that both *x* and *someValue* refer to the same underlying data and modifications to either are visible to both. Algorithm steps that want to avoid this reference-like behaviour should explicitly make a copy of the right-hand side: “Let *x* be a copy of *someValue*” creates a shallow copy of *someValue*.

Once declared, an alias may be referenced in any subsequent steps and must not be referenced from steps prior to the alias's declaration. Aliases may be modified using the form “Set *x* to *someOtherValue*”.

5.2.1 Abstract Operations

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. Abstract operations are typically referenced using a functional application style such as `OperationName(arg1, arg2)`. Some abstract operations are treated as polymorphically dispatched methods of class-like specification abstractions. Such method-like abstract operations are typically referenced using a method application style such as `someValue.OperationName(arg1, arg2)`.

5.2.2 Syntax-Directed Operations

A *syntax-directed operation* is a named operation whose definition consists of algorithms, each of which is associated with one or more productions from 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 source text. The *source text matched by* a grammar production is the portion of the source text that starts at the beginning of the first terminal that participated in the match and ends at the end of the last terminal that participated in the match.

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.

Syntax-directed operations are invoked with a parse node and, optionally, other parameters by using the conventions on steps 1, 3, and 4 in the following algorithm:

1. Let *status* be SyntaxDirectedOperation of *SomeNonTerminal*.
2. Let *someParseNode* be the parse of some source text.
3. Perform SyntaxDirectedOperation of *someParseNode*.
4. Perform SyntaxDirectedOperation of *someParseNode* passing "value" as the argument.

Unless explicitly specified otherwise, all chain productions have an implicit definition for every operation that might be applied to that production's left-hand side nonterminal. The implicit definition simply reapplies the same operation with the same parameters, if any, to the [chain production](#)'s sole right-hand side nonterminal and then returns the result. For example, assume that some algorithm has a step of the form: “Return the result of evaluating *Block*” and that there is a production:

Block :
 { *StatementList* }

but the Evaluation operation does not associate an algorithm with that production. In that case, the Evaluation operation implicitly includes an association of the form:

Runtime Semantics: Evaluation

Block : { *StatementList* }

1. Return the result of evaluating *StatementList*.

5.2.3 Runtime Semantics

Algorithms which specify semantics that must be called at runtime are called *runtime semantics*. Runtime semantics are defined by [abstract operations](#) or syntax-directed operations. Such algorithms always return a completion record.

5.2.3.1 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".

means the same thing as:

1. Return [NormalCompletion](#)("Infinity").

However, if the value expression of a “return” statement is a **Completion Record** construction literal, the resulting **Completion Record** is returned. If the value expression is a call to an abstract operation, the “return” statement simply returns the **Completion Record** produced by the abstract operation.

The abstract operation **Completion**(*completionRecord*) is used to emphasize that a previously computed **Completion Record** is being returned. The **Completion** abstract operation takes a single argument, *completionRecord*, and performs the following steps:

1. **Assert**: *completionRecord* is a **Completion Record**.
2. Return *completionRecord* as the **Completion Record** of this abstract operation.

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

1. Return **NormalCompletion**(**undefined**).

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

5.2.3.2 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 **ThrowCompletion**(a newly created **TypeError** object).

5.2.3.3 ReturnIfAbrupt

Algorithms steps that say or are otherwise equivalent to:

1. **ReturnIfAbrupt**(*argument*).

mean the same thing as:

1. If *argument* is an abrupt completion, return *argument*.
2. Else if *argument* is a **Completion Record**, set *argument* to *argument*.**[[Value]]**.

Algorithms steps that say or are otherwise equivalent to:

1. **ReturnIfAbrupt**(**AbstractOperation**()).

mean the same thing as:

1. Let *hygienicTemp* be **AbstractOperation**()
2. If *hygienicTemp* is an abrupt completion, return *hygienicTemp*.
3. Else if *hygienicTemp* is a **Completion Record**, set *hygienicTemp* to *hygienicTemp*.**[[Value]]**.

Where *hygienicTemp* is ephemeral and visible only in the steps pertaining to **ReturnIfAbrupt**.

Algorithms steps that say or are otherwise equivalent to:

1. Let *result* be `AbstractOperation(ReturnIfAbrupt(argument))`.

mean the same thing as:

1. If *argument* is an `abrupt completion`, return *argument*.
2. If *argument* is a `Completion Record`, set *argument* to *argument*.[[Value]].
3. Let *result* be `AbstractOperation(argument)`.

5.2.3.4 ReturnIfAbrupt Shorthands

Invocations of `abstract operations` and syntax-directed operations that are prefixed by `?` indicate that `ReturnIfAbrupt` should be applied to the resulting `Completion Record`. For example, the step:

1. `? OperationName()`.

is equivalent to the following step:

1. `ReturnIfAbrupt(OperationName())`.

Similarly, for method application style, the step:

1. `? someValue.OperationName()`.

is equivalent to:

1. `ReturnIfAbrupt(someValue.OperationName())`.

Similarly, prefix `!` is used to indicate that the following invocation of an abstract or syntax-directed operation will never return an `abrupt completion` and that the resulting `Completion Record`'s [[Value]] field should be used in place of the return value of the operation. For example, the step:

1. Let *val* be `! OperationName()`.

is equivalent to the following steps:

1. Let *val* be `OperationName()`.
2. `Assert: val` is never an `abrupt completion`.
3. If *val* is a `Completion Record`, set *val* to *val*.[[Value]].

Syntax-directed operations for `runtime semantics` make use of this shorthand by placing `!` or `?` before the invocation of the operation:

1. Perform `! SyntaxDirectedOperation of NonTerminal`.

5.2.4 Static Semantics

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* or *Module* 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.

A special kind of static semantic rule is an *Early Error Rule*. **Early error** rules define **early error** conditions (see clause 17) 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* or *Module*, validate all of the **early error** rules of the productions used to parse that *Script* or *Module*. If any of the **early error** rules are violated the *Script* or *Module* is invalid and cannot be evaluated.

5.2.5 Mathematical Operations

This specification makes reference to these kinds of numeric values:

- *Mathematical values*: Arbitrary real numbers, used as the default numeric type.
- *Extended mathematical values*: Mathematical values together with $+\infty$ and $-\infty$.
- *Numbers*: [IEEE 754-2019](#) double-precision floating point values.
- *BigInts*: ECMAScript values representing arbitrary integers in a one-to-one correspondence.

In the language of this specification, numerical values are distinguished among different numeric kinds using subscript suffixes. The subscript \mathbb{F} refers to Numbers, and the subscript \mathbb{Z} refers to BigInts. Numeric values without a subscript suffix refer to mathematical values.

Numeric operators such as $+$, \times , $=$, and \geq refer to those operations as determined by the type of the operands. When applied to mathematical values, the operators refer to the usual mathematical operations. When applied to Numbers, the operators refer to the relevant operations within [IEEE 754-2019](#). When applied to BigInts, the operators refer to the usual mathematical operations applied to the **mathematical value** of the BigInt.

In general, when this specification refers to a numerical value, such as in the phrase, "the length of y " or "the **integer** represented by the four hexadecimal digits ...", without explicitly specifying a numeric kind, the phrase refers to a **mathematical value**. Phrases which refer to a Number or a BigInt value are explicitly annotated as such; for example, "the **Number value** for the number of code points in ..." or "the **BigInt value** for ...".

Numeric operators applied to mixed-type operands (such as a Number and a **mathematical value**) are not defined and should be considered an editorial error in this specification.

This specification denotes most numeric values in base 10; it also uses numeric values of the form 0x followed by digits 0-9 or A-F as base-16 values.

When the term *integer* is used in this specification, it refers to a **mathematical value** which is in the set of integers, unless otherwise stated. When the term *integral Number* is used in this specification, it refers to a **Number value** whose **mathematical value** is in the set of integers.

Conversions between mathematical values and Numbers or BigInts are always explicit in this document. A conversion from a **mathematical value** or **extended mathematical value** x to a Number is denoted as "the **Number value** for x " or $\mathbb{F}(x)$, and is defined in 6.1.6.1. A conversion from an **integer** x to a BigInt is denoted as "the **BigInt value** for x " or $\mathbb{Z}(x)$. A conversion from a Number or BigInt x to a **mathematical value** is denoted as "the *mathematical value* of x ", or $\mathbb{R}(x)$. The **mathematical value** of $+0_{\mathbb{F}}$ and $-0_{\mathbb{F}}$ is the **mathematical value** 0. The **mathematical value** of non-finite values is not defined. The *extended mathematical value* of x is the **mathematical value** of x for finite values, and is $+\infty$ and $-\infty$ for $+\infty_{\mathbb{F}}$ and $-\infty_{\mathbb{F}}$ respectively; it is not defined for NaN.

The mathematical function $\text{abs}(x)$ produces the absolute value of x , which is $-x$ if $x < 0$ and otherwise is x itself.

The mathematical function $\min(x_1, x_2, \dots, x_N)$ produces the mathematically smallest of x_1 through x_N . The mathematical function $\max(x_1, x_2, \dots, x_N)$ produces the mathematically largest of x_1 through x_N . The domain and range of these mathematical functions are the extended mathematical values.

The notation “ x modulo y ” (y must be finite and non-zero) 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 phrase “the result of *clamping* x between *lower* and *upper*” (where x is an extended mathematical value and *lower* and *upper* are mathematical values such that $\text{lower} \leq \text{upper}$) produces *lower* if $x < \text{lower}$, produces *upper* if $x > \text{upper}$, and otherwise produces x .

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

Mathematical functions \min , \max , abs , and floor are not defined for Numbers and BigInts, and any usage of those methods that have non-mathematical value arguments would be an editorial error in this specification.

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

5.2.6 Value Notation

In this specification, ECMAScript language values are displayed in **bold**. Examples include **null**, **true**, or **"hello"**. These are distinguished from longer ECMAScript code sequences such as **Function.prototype.apply** or **let n = 42;**.

Values which are internal to the specification and not directly observable from ECMAScript code are indicated with a sans-serif typeface. For instance, a [Completion Record](#)'s `[[Type]]` field takes on values like `normal`, `return`, or `throw`.

6 ECMAScript Data Types and Values

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

Within this specification, the notation “ $\text{Type}(x)$ ” is used as shorthand for “the *type* of x ” where “*type*” refers to the ECMAScript language and specification types defined in this clause. When the term “empty” is used as if it was naming a value, it is equivalent to saying “no value of any type”.

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, BigInt, 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 ordered sequences of zero or more 16-bit unsigned [integer](#) values (“elements”) up to a maximum length of $2^{53} - 1$ 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 non-negative 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.

ECMAScript operations that do not interpret String contents apply no further semantics. Operations that do interpret String values treat each element as a single UTF-16 code unit. However, ECMAScript does not restrict the value of or relationships between these code units, so operations that further interpret String contents as sequences of Unicode code points encoded in UTF-16 must account for ill-formed subsequences. Such operations apply special treatment to every code unit with a numeric value in the inclusive range 0xD800 to 0xDBFF (defined by the Unicode Standard as a *leading surrogate*, or more formally as a *high-surrogate code unit*) and every code unit with a numeric value in the inclusive range 0xDC00 to 0xDFFF (defined as a *trailing surrogate*, or more formally as a *low-surrogate code unit*) using the following rules:

- A code unit that is not a [leading surrogate](#) and not a [trailing surrogate](#) is interpreted as a code point with the same value.
- A sequence of two code units, where the first code unit *c1* is a [leading surrogate](#) and the second code unit *c2* a [trailing surrogate](#), is a *surrogate pair* and is interpreted as a code point with the value $(c1 - 0xD800) \times 0x400 + (c2 - 0xDC00) + 0x10000$. (See [11.1.3](#))
- A code unit that is a [leading surrogate](#) or [trailing surrogate](#), but is not part of a [surrogate pair](#), is interpreted as a code point with the same value.

The function **String.prototype.normalize** (see [22.1.3.13](#)) can be used to explicitly normalize a String value. **String.prototype.localeCompare** (see [22.1.3.10](#)) internally normalizes String values, but no other operations implicitly normalize the strings upon which they operate. 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 text is in Normalized Form C, string literals are guaranteed to also be normalized, as long as they do not contain any Unicode escape sequences.

In this specification, the phrase “the *string-concatenation* of *A*, *B*, ...” (where each argument is a String value, a code unit,

or a sequence of code units) denotes the String value whose sequence of code units is the concatenation of the code units (in order) of each of the arguments (in order).

The phrase "the *substring* of *S* from *inclusiveStart* to *exclusiveEnd*" (where *S* is a String value or a sequence of code units and *inclusiveStart* and *exclusiveEnd* are integers) denotes the String value consisting of the consecutive code units of *S* beginning at index *inclusiveStart* and ending immediately before index *exclusiveEnd* (which is the empty String when *inclusiveStart* = *exclusiveEnd*). If the "to" suffix is omitted, the length of *S* is used as the value of *exclusiveEnd*.

6.1.4.1 StringIndexOf (*string*, *searchValue*, *fromIndex*)

The abstract operation StringIndexOf takes arguments *string* (a String), *searchValue* (a String), and *fromIndex* (a non-negative integer). It performs the following steps when called:

1. **Assert:** Type(*string*) is String.
2. **Assert:** Type(*searchValue*) is String.
3. **Assert:** *fromIndex* is a non-negative integer.
4. Let *len* be the length of *string*.
5. If *searchValue* is the empty String and *fromIndex* ≤ *len*, return *fromIndex*.
6. Let *searchLen* be the length of *searchValue*.
7. For each integer *i* starting with *fromIndex* such that $i \leq len - searchLen$, in ascending order, do
 - a. Let *candidate* be the **substring** of *string* from *i* to $i + searchLen$.
 - b. If *candidate* is the same sequence of code units as *searchValue*, return *i*.
8. Return -1.

NOTE 1 If *searchValue* is the empty String and *fromIndex* is less than or equal to the length of *string*, this algorithm returns *fromIndex*. The empty String is effectively found at every position within a string, including after the last code unit.

NOTE 2 This algorithm always returns -1 if *fromIndex* > the length of *string*.

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 value 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 realms (9.2).

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

Table 1: Well-known Symbols

Specification Name	[[Description]]	Value and Purpose
@@asyncIterator	"Symbol.asyncIterator"	A method that returns the default AsyncIterator for an object. Called by the semantics of the for-await-of statement.
@@hasInstance	"Symbol.hasInstance"	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.
@@isConcatSpreadable	"Symbol.isConcatSpreadable"	A Boolean valued property that if true indicates that an object should be flattened to its array elements by Array.prototype.concat .
@@iterator	"Symbol.iterator"	A method that returns the default Iterator for an object. Called by the semantics of the for-of statement.
@@match	"Symbol.match"	A regular expression method that matches the regular expression against a string. Called by the String.prototype.match method.
@@matchAll	"Symbol.matchAll"	A regular expression method that returns an iterator, that yields matches of the regular expression against a string. Called by the String.prototype.matchAll method.
@@replace	"Symbol.replace"	A regular expression method that replaces matched substrings of a string. Called by the String.prototype.replace method.
@@search	"Symbol.search"	A regular expression method that returns the index within a string that matches the regular expression. Called by the String.prototype.search method.
@@species	"Symbol.species"	A function valued property that is the constructor function that is used to create derived objects.
@@split	"Symbol.split"	A regular expression method that splits a string at the indices that match the regular expression. Called by the String.prototype.split method.
@@toPrimitive	"Symbol.toPrimitive"	A method that converts an object to a corresponding primitive value. Called by the ToPrimitive abstract operation.
@@toStringTag	"Symbol.toStringTag"	A String valued property that is used in the creation of the default string description of an object. Accessed by the built-in method Object.prototype.toString .
@@unscopables	"Symbol.unscopables"	An object valued property whose own and inherited property names are property names that are excluded from the with environment bindings of the associated object.

6.1.6 Numeric Types

ECMAScript has two built-in numeric types: Number and BigInt. In this specification, every numeric type T contains a multiplicative identity value denoted $T::\text{unit}$. The specification types also have the following [abstract operations](#), likewise denoted $T::op$ for a given operation with specification name op . All argument types are T . The "Result" column shows the return type, along with an indication if it is possible for some invocations of the operation to return an [abrupt completion](#).

Table 2: Numeric Type Operations

Invocation Synopsis	Example source	Invoked by the Evaluation semantics of ...	Result
$T::\text{unaryMinus}(x)$	$-x$	Unary - Operator	T
$T::\text{bitwiseNOT}(x)$	$\sim x$	Bitwise NOT Operator (\sim)	T
$T::\text{exponentiate}(x, y)$	$x ** y$	Exponentiation Operator and <code>Math.pow (base, exponent)</code>	T , may throw RangeError
$T::\text{multiply}(x, y)$	$x * y$	Multiplicative Operators	T
$T::\text{divide}(x, y)$	x / y	Multiplicative Operators	T , may throw RangeError
$T::\text{remainder}(x, y)$	$x \% y$	Multiplicative Operators	T , may throw RangeError
$T::\text{add}(x, y)$	$x ++$ $++ x$ $x + y$	Postfix Increment Operator, Prefix Increment Operator, and The Addition Operator ($+$)	T
$T::\text{subtract}(x, y)$	$x --$ $-- x$ $x - y$	Postfix Decrement Operator, Prefix Decrement Operator, and The Subtraction Operator ($-$)	T
$T::\text{leftShift}(x, y)$	$x \ll y$	The Left Shift Operator (\ll)	T
$T::\text{signedRightShift}(x, y)$	$x \gg y$	The Signed Right Shift Operator (\gg)	T
$T::\text{unsignedRightShift}(x, y)$	$x \ggg y$	The Unsigned Right Shift Operator (\ggg)	T , may throw TypeError
$T::\text{lessThan}(x, y)$	$x < y$ $x > y$ $x \leq y$ $x \geq y$	Relational Operators, via Abstract Relational Comparison	Boolean or undefined (for unordered inputs)
$T::\text{equal}(x, y)$	$x == y$ $x !== y$ $x === y$ $x !== y$	Equality Operators, via Strict Equality Comparison	Boolean
$T::\text{sameValue}(x, y)$		Object internal methods, via <code>SameValue (x, y)</code> , to test	Boolean

		exact value equality	
<code>T::sameValueZero(x, y)</code>		Array, Map, and Set methods, via <code>SameValueZero (x, y)</code> , to test value equality ignoring differences among members of the zero cohort (i.e., <code>-0_F</code> and <code>+0_F</code>)	Boolean
<code>T::bitwiseAND(x, y)</code>	<code>x & y</code>	Binary Bitwise Operators	<i>T</i>
<code>T::bitwiseXOR(x, y)</code>	<code>x ^ y</code>	Binary Bitwise Operators	<i>T</i>
<code>T::bitwiseOR(x, y)</code>	<code>x y</code>	Binary Bitwise Operators	<i>T</i>
<code>T::toString(x)</code>	String(x)	Many expressions and built-in functions, via <code>ToString (argument)</code>	String

The `T::unit` value and `T::op` 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 [abstract operations](#) are defined throughout this specification.

Because the numeric types are in general not convertible without loss of precision or truncation, the ECMAScript language provides no implicit conversion among these types. Programmers must explicitly call **Number** and **BigInt** functions to convert among types when calling a function which requires another type.

NOTE The first and subsequent editions of ECMAScript have provided, for certain operators, implicit numeric conversions that could lose precision or truncate. These legacy implicit conversions are maintained for backward compatibility, but not provided for **BigInt** in order to minimize opportunity for programmer error, and to leave open the option of generalized *value types* in a future edition.

6.1.6.1 The Number Type

The **Number** type has exactly 18,437,736,874,454,810,627 (that is, $2^{64} - 2^{53} + 3$) values, representing the double-precision 64-bit format [IEEE 754-2019](#) values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the 9,007,199,254,740,990 (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-defined](#); to ECMAScript code, all **NaN** values are indistinguishable from each other.

NOTE The bit pattern that might be observed in an `ArrayBuffer` (see [25.1](#)) or a `SharedArrayBuffer` (see [25.2](#)) after a **Number value** has been stored into it is not necessarily the same as the internal representation of that **Number value** used by the ECMAScript implementation.

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 `+∞F` and `-∞F`, respectively. (Note that these two infinite **Number** values are produced by the program expressions **+Infinity** (or simply **Infinity**) and **-Infinity**.)

The other 18,437,736,874,454,810,624 (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 $+0_{\mathbb{F}}$ and $-0_{\mathbb{F}}$, respectively. (Note that these two different zero **Number values** are produced by the program expressions $+0$ (or simply 0) and -0 .)

The 18,437,736,874,454,810,622 (that is, $2^{64} - 2^{53} - 2$) finite non-zero values are of two kinds:

18,428,729,675,200,069,632 (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 an **integer** such that $2^{52} \leq m < 2^{53}$, and e is an **integer** such that $-1074 \leq e \leq 971$.

The remaining 9,007,199,254,740,990 (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 an **integer** such that $0 < m < 2^{52}$, 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. The **integer** 0 has two representations in the **Number** type: $+0_{\mathbb{F}}$ and $-0_{\mathbb{F}}$.

A finite number has an *odd significand* if it is non-zero 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 real mathematical quantity (which might even be an irrational number such as π) means a **Number value** chosen in the following manner. Consider the set of all finite values of the **Number** type, with $-0_{\mathbb{F}}$ removed and with two additional values added to it that are not representable in the **Number** type, namely 2^{1024} (which is $+1 \times 2^{53} \times 2^{971}$) and -2^{1024} (which is $-1 \times 2^{53} \times 2^{971}$). Choose the member of this set that is closest in value to x . If two values of the set are equally close, then the one with an even significand is chosen; for this purpose, the two extra values 2^{1024} and -2^{1024} are considered to have even significands. Finally, if 2^{1024} was chosen, replace it with $+\infty_{\mathbb{F}}$; if -2^{1024} was chosen, replace it with $-\infty_{\mathbb{F}}$; if $+0_{\mathbb{F}}$ was chosen, replace it with $-0_{\mathbb{F}}$ if and only if $x < 0$; 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 754-2019](#) roundTiesToEven mode.)

The **Number value** for $+\infty$ is $+\infty_{\mathbb{F}}$, and the **Number value** for $-\infty$ is $-\infty_{\mathbb{F}}$.

Some ECMAScript operators deal only with integers in specific ranges such as -2^{31} through $2^{31} - 1$, inclusive, or in the range 0 through $2^{16} - 1$, inclusive. These operators accept any value of the **Number** type but first convert each such value to an **integer** value in the expected range. See the descriptions of the numeric conversion operations in [7.1](#).

The **Number::unit** value is $1_{\mathbb{F}}$.

6.1.6.1.1 **Number::unaryMinus** (x)

The abstract operation **Number::unaryMinus** takes argument x (a **Number**). It performs the following steps when called:

1. If x is NaN, return NaN.
2. Return the result of negating x ; that is, compute a Number with the same magnitude but opposite sign.

6.1.6.1.2 Number::bitwiseNOT (x)

The abstract operation Number::bitwiseNOT takes argument x (a Number). It performs the following steps when called:

1. Let *oldValue* be ! ToInt32(x).
2. Return the result of applying bitwise complement to *oldValue*. The mathematical value of the result is exactly representable as a 32-bit two's complement bit string.

6.1.6.1.3 Number::exponentiate ($base$, $exponent$)

The abstract operation Number::exponentiate takes arguments $base$ (a Number) and $exponent$ (a Number). It returns an implementation-approximated value representing the result of raising $base$ to the $exponent$ power. It performs the following steps when called:

1. If $exponent$ is NaN, return NaN.
2. If $exponent$ is $+0_{\mathbb{F}}$ or $exponent$ is $-0_{\mathbb{F}}$, return $1_{\mathbb{F}}$.
3. If $base$ is NaN, return NaN.
4. If $base$ is $+\infty_{\mathbb{F}}$, then
 - a. If $exponent > +0_{\mathbb{F}}$, return $+\infty_{\mathbb{F}}$. Otherwise, return $+0_{\mathbb{F}}$.
5. If $base$ is $-\infty_{\mathbb{F}}$, then
 - a. If $exponent > +0_{\mathbb{F}}$, then
 - i. If $exponent$ is an odd integral Number, return $-\infty_{\mathbb{F}}$. Otherwise, return $+\infty_{\mathbb{F}}$.
 - b. Else,
 - i. If $exponent$ is an odd integral Number, return $-0_{\mathbb{F}}$. Otherwise, return $+0_{\mathbb{F}}$.
6. If $base$ is $+0_{\mathbb{F}}$, then
 - a. If $exponent > +0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$. Otherwise, return $+\infty_{\mathbb{F}}$.
7. If $base$ is $-0_{\mathbb{F}}$, then
 - a. If $exponent > +0_{\mathbb{F}}$, then
 - i. If $exponent$ is an odd integral Number, return $-0_{\mathbb{F}}$. Otherwise, return $+0_{\mathbb{F}}$.
 - b. Else,
 - i. If $exponent$ is an odd integral Number, return $-\infty_{\mathbb{F}}$. Otherwise, return $+\infty_{\mathbb{F}}$.
8. Assert: $base$ is finite and is neither $+0_{\mathbb{F}}$ nor $-0_{\mathbb{F}}$.
9. If $exponent$ is $+\infty_{\mathbb{F}}$, then
 - a. If $\text{abs}(\mathbb{R}(base)) > 1$, return $+\infty_{\mathbb{F}}$.
 - b. If $\text{abs}(\mathbb{R}(base))$ is 1, return NaN.
 - c. If $\text{abs}(\mathbb{R}(base)) < 1$, return $+0_{\mathbb{F}}$.
10. If $exponent$ is $-\infty_{\mathbb{F}}$, then
 - a. If $\text{abs}(\mathbb{R}(base)) > 1$, return $+0_{\mathbb{F}}$.
 - b. If $\text{abs}(\mathbb{R}(base))$ is 1, return NaN.
 - c. If $\text{abs}(\mathbb{R}(base)) < 1$, return $+\infty_{\mathbb{F}}$.
11. Assert: $exponent$ is finite and is neither $+0_{\mathbb{F}}$ nor $-0_{\mathbb{F}}$.
12. If $base < +0_{\mathbb{F}}$ and $exponent$ is not an integral Number, return NaN.
13. Return an implementation-approximated value representing the result of raising $\mathbb{R}(base)$ to the $\mathbb{R}(exponent)$

power.

NOTE The result of *base* ****** *exponent* when *base* is $1_{\mathbb{F}}$ or $-1_{\mathbb{F}}$ and *exponent* is $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, or when *base* is $1_{\mathbb{F}}$ and *exponent* is NaN, differs from IEEE 754-2019. The first edition of ECMAScript specified a result of NaN for this operation, whereas later versions of IEEE 754-2019 specified $1_{\mathbb{F}}$. The historical ECMAScript behaviour is preserved for compatibility reasons.

6.1.6.1.4 Number::multiply (*x*, *y*)

The abstract operation Number::multiply takes arguments *x* (a Number) and *y* (a Number). It performs multiplication according to the rules of IEEE 754-2019 binary double-precision arithmetic, producing the product of *x* and *y*. It performs the following steps when called:

1. If *x* is NaN or *y* is NaN, return NaN.
2. If *x* is $+\infty_{\mathbb{F}}$ or *x* is $-\infty_{\mathbb{F}}$, then
 - a. If *y* is $+0_{\mathbb{F}}$ or *y* is $-0_{\mathbb{F}}$, return NaN.
 - b. If *y* $> +0_{\mathbb{F}}$, return *x*.
 - c. Return $-x$.
3. If *y* is $+\infty_{\mathbb{F}}$ or *y* is $-\infty_{\mathbb{F}}$, then
 - a. If *x* is $+0_{\mathbb{F}}$ or *x* is $-0_{\mathbb{F}}$, return NaN.
 - b. If *x* $> +0_{\mathbb{F}}$, return *y*.
 - c. Return $-y$.
4. Return $\mathbb{F}(\mathbb{R}(x) \times \mathbb{R}(y))$.

NOTE Finite-precision multiplication is commutative, but not always associative.

6.1.6.1.5 Number::divide (*x*, *y*)

The abstract operation Number::divide takes arguments *x* (a Number) and *y* (a Number). It performs division according to the rules of IEEE 754-2019 binary double-precision arithmetic, producing the quotient of *x* and *y* where *x* is the dividend and *y* is the divisor. It performs the following steps when called:

1. If *x* is NaN or *y* is NaN, return NaN.
2. If *x* is $+\infty_{\mathbb{F}}$ or *x* is $-\infty_{\mathbb{F}}$, then
 - a. If *y* is $+\infty_{\mathbb{F}}$ or *y* is $-\infty_{\mathbb{F}}$, return NaN.
 - b. If *y* is $+0_{\mathbb{F}}$ or *y* $> +0_{\mathbb{F}}$, return *x*.
 - c. Return $-x$.
3. If *y* is $+\infty_{\mathbb{F}}$, then
 - a. If *x* is $+0_{\mathbb{F}}$ or *x* $> +0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$. Otherwise, return $-0_{\mathbb{F}}$.
4. If *y* is $-\infty_{\mathbb{F}}$, then
 - a. If *x* is $+0_{\mathbb{F}}$ or *x* $> +0_{\mathbb{F}}$, return $-0_{\mathbb{F}}$. Otherwise, return $+0_{\mathbb{F}}$.
5. If *x* is $+0_{\mathbb{F}}$ or *x* is $-0_{\mathbb{F}}$, then
 - a. If *y* is $+0_{\mathbb{F}}$ or *y* is $-0_{\mathbb{F}}$, return NaN.
 - b. If *y* $> +0_{\mathbb{F}}$, return *x*.
 - c. Return $-x$.
6. If *y* is $+0_{\mathbb{F}}$, then

- a. If $x > +0_{\mathbb{F}}$, return $+\infty_{\mathbb{F}}$. Otherwise, return $-\infty_{\mathbb{F}}$.
7. If y is $-0_{\mathbb{F}}$ then
 - a. If $x > +0_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$. Otherwise, return $+\infty_{\mathbb{F}}$.
8. Return $\mathbb{F}(\mathbb{R}(x) / \mathbb{R}(y))$.

6.1.6.1.6 Number::remainder (n, d)

The abstract operation Number::remainder takes arguments n (a Number) and d (a Number). It yields the remainder from an implied division of its operands where n is the dividend and d is the divisor. It performs the following steps when called:

1. If n is NaN or d is NaN, return NaN.
2. If n is $+\infty_{\mathbb{F}}$ or n is $-\infty_{\mathbb{F}}$, return NaN.
3. If d is $+\infty_{\mathbb{F}}$ or d is $-\infty_{\mathbb{F}}$, return n .
4. If d is $+0_{\mathbb{F}}$ or d is $-0_{\mathbb{F}}$, return NaN.
5. If n is $+0_{\mathbb{F}}$ or n is $-0_{\mathbb{F}}$, return n .
6. **Assert**: n and d are finite and non-zero.
7. Let r be $\mathbb{R}(n) - (\mathbb{R}(d) \times q)$ where q is an integer that is negative if and only if n and d have opposite sign, and whose magnitude is as large as possible without exceeding the magnitude of $\mathbb{R}(n) / \mathbb{R}(d)$.
8. Return $\mathbb{F}(r)$.

NOTE 1 In C and C++, the remainder operator accepts only integral operands; in ECMAScript, it also accepts floating-point operands.

NOTE 2 The result of a floating-point remainder operation as computed by the **%** operator is not the same as the “remainder” operation defined by IEEE 754-2019. The IEEE 754-2019 “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.

6.1.6.1.7 Number::add (x, y)

The abstract operation Number::add takes arguments x (a Number) and y (a Number). It performs addition according to the rules of IEEE 754-2019 binary double-precision arithmetic, producing the sum of its arguments. It performs the following steps when called:

1. If x is NaN or y is NaN, return NaN.
2. If x is $+\infty_{\mathbb{F}}$ and y is $-\infty_{\mathbb{F}}$, return NaN.
3. If x is $-\infty_{\mathbb{F}}$ and y is $+\infty_{\mathbb{F}}$, return NaN.
4. If x is $+\infty_{\mathbb{F}}$ or x is $-\infty_{\mathbb{F}}$, return x .
5. If y is $+\infty_{\mathbb{F}}$ or y is $-\infty_{\mathbb{F}}$, return y .
6. **Assert**: x and y are both finite.
7. If x is $-0_{\mathbb{F}}$ and y is $-0_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
8. Return $\mathbb{F}(\mathbb{R}(x) + \mathbb{R}(y))$.

6.1.6.1.8 Number::subtract (x, y)

The abstract operation `Number::subtract` takes arguments x (a Number) and y (a Number). It performs subtraction, producing the difference of its operands; x is the minuend and y is the subtrahend. It performs the following steps when called:

1. Return `Number::add(x, Number::unaryMinus(y))`.

6.1.6.1.9 Number::leftShift (x, y)

The abstract operation `Number::leftShift` takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. Let $lnum$ be `! ToInt32(x)`.
2. Let $rnum$ be `! ToUint32(y)`.
3. Let $shiftCount$ be `$\mathbb{R}(rnum)$ modulo 32`.
4. Return the result of left shifting $lnum$ by $shiftCount$ bits. The **mathematical value** of the result is exactly representable as a 32-bit two's complement bit string.

6.1.6.1.10 Number::signedRightShift (x, y)

The abstract operation `Number::signedRightShift` takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. Let $lnum$ be `! ToInt32(x)`.
2. Let $rnum$ be `! ToUint32(y)`.
3. Let $shiftCount$ be `$\mathbb{R}(rnum)$ modulo 32`.
4. Return the result of performing a sign-extending right shift of $lnum$ by $shiftCount$ bits. The most significant bit is propagated. The **mathematical value** of the result is exactly representable as a 32-bit two's complement bit string.

6.1.6.1.11 Number::unsignedRightShift (x, y)

The abstract operation `Number::unsignedRightShift` takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. Let $lnum$ be `! ToUint32(x)`.
2. Let $rnum$ be `! ToUint32(y)`.
3. Let $shiftCount$ be `$\mathbb{R}(rnum)$ modulo 32`.
4. Return the result of performing a zero-filling right shift of $lnum$ by $shiftCount$ bits. Vacated bits are filled with zero. The **mathematical value** of the result is exactly representable as a 32-bit unsigned bit string.

6.1.6.1.12 Number::lessThan (x, y)

The abstract operation `Number::lessThan` takes arguments x (a Number) and y (a Number). It performs the following

steps when called:

1. If x is **NaN**, return **undefined**.
2. If y is **NaN**, return **undefined**.
3. If x and y are the same **Number value**, return **false**.
4. If x is $+0_{\mathbb{F}}$ and y is $-0_{\mathbb{F}}$, return **false**.
5. If x is $-0_{\mathbb{F}}$ and y is $+0_{\mathbb{F}}$, return **false**.
6. If x is $+\infty_{\mathbb{F}}$, return **false**.
7. If y is $+\infty_{\mathbb{F}}$, return **true**.
8. If y is $-\infty_{\mathbb{F}}$, return **false**.
9. If x is $-\infty_{\mathbb{F}}$, return **true**.
10. **Assert**: x and y are finite and non-zero.
11. If $\mathbb{R}(x) < \mathbb{R}(y)$, return **true**. Otherwise, return **false**.

6.1.6.1.13 **Number::equal** (x, y)

The abstract operation **Number::equal** takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. If x is **NaN**, return **false**.
2. If y is **NaN**, return **false**.
3. If x is the same **Number value** as y , return **true**.
4. If x is $+0_{\mathbb{F}}$ and y is $-0_{\mathbb{F}}$, return **true**.
5. If x is $-0_{\mathbb{F}}$ and y is $+0_{\mathbb{F}}$, return **true**.
6. Return **false**.

6.1.6.1.14 **Number::sameValue** (x, y)

The abstract operation **Number::sameValue** takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. If x is **NaN** and y is **NaN**, return **true**.
2. If x is $+0_{\mathbb{F}}$ and y is $-0_{\mathbb{F}}$, return **false**.
3. If x is $-0_{\mathbb{F}}$ and y is $+0_{\mathbb{F}}$, return **false**.
4. If x is the same **Number value** as y , return **true**.
5. Return **false**.

6.1.6.1.15 **Number::sameValueZero** (x, y)

The abstract operation **Number::sameValueZero** takes arguments x (a Number) and y (a Number). It performs the following steps when called:

1. If x is **NaN** and y is **NaN**, return **true**.
2. If x is $+0_{\mathbb{F}}$ and y is $-0_{\mathbb{F}}$, return **true**.
3. If x is $-0_{\mathbb{F}}$ and y is $+0_{\mathbb{F}}$, return **true**.
4. If x is the same **Number value** as y , return **true**.
5. Return **false**.

6.1.6.1.16 **NumberBitwiseOp** (op, x, y)

The abstract operation `NumberBitwiseOp` takes arguments *op* (a sequence of Unicode code points), *x*, and *y*. It performs the following steps when called:

1. **Assert**: *op* is **&**, **^**, or **|**.
2. Let *lnum* be ! `ToInt32(x)`.
3. Let *rnum* be ! `ToInt32(y)`.
4. Let *lbits* be the 32-bit two's complement bit string representing $\mathbb{R}(lnum)$.
5. Let *rbits* be the 32-bit two's complement bit string representing $\mathbb{R}(rnum)$.
6. If *op* is **&**, let *result* be the result of applying the bitwise AND operation to *lbits* and *rbits*.
7. Else if *op* is **^**, let *result* be the result of applying the bitwise exclusive OR (XOR) operation to *lbits* and *rbits*.
8. Else, *op* is **|**. Let *result* be the result of applying the bitwise inclusive OR operation to *lbits* and *rbits*.
9. Return the **Number value** for the **integer** represented by the 32-bit two's complement bit string *result*.

6.1.6.1.17 `Number::bitwiseAND (x, y)`

The abstract operation `Number::bitwiseAND` takes arguments *x* (a Number) and *y* (a Number). It performs the following steps when called:

1. Return `NumberBitwiseOp(&, x, y)`.

6.1.6.1.18 `Number::bitwiseXOR (x, y)`

The abstract operation `Number::bitwiseXOR` takes arguments *x* (a Number) and *y* (a Number). It performs the following steps when called:

1. Return `NumberBitwiseOp(^, x, y)`.

6.1.6.1.19 `Number::bitwiseOR (x, y)`

The abstract operation `Number::bitwiseOR` takes arguments *x* (a Number) and *y* (a Number). It performs the following steps when called:

1. Return `NumberBitwiseOp(|, x, y)`.

6.1.6.1.20 `Number::toString (x)`

The abstract operation `Number::toString` takes argument *x* (a Number). It converts *x* to String format. It performs the following steps when called:

1. If *x* is **NaN**, return the String **"NaN"**.
2. If *x* is **+0_F** or **-0_F**, return the String **"0"**.
3. If *x* < **+0_F**, return the **string-concatenation** of **"-"** and ! `Number::toString(-x)`.
4. If *x* is **+∞_F**, return the String **"Infinity"**.
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, $s \times 10^{n-k}$ is $\mathbb{R}(x)$, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the **string-concatenation** of:
 - the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes)
 - $n - k$ occurrences of the code unit 0x0030 (DIGIT ZERO)
7. If $0 < n \leq 21$, return the **string-concatenation** of:

- the code units of the most significant n digits of the decimal representation of s
 - the code unit 0x002E (FULL STOP)
 - the code units of the remaining $k - n$ digits of the decimal representation of s
8. If $-6 < n \leq 0$, return the **string-concatenation** of:
- the code unit 0x0030 (DIGIT ZERO)
 - the code unit 0x002E (FULL STOP)
 - $-n$ occurrences of the code unit 0x0030 (DIGIT ZERO)
 - the code units of the k digits of the decimal representation of s
9. Otherwise, if $k = 1$, return the **string-concatenation** of:
- the code unit of the single digit of s
 - the code unit 0x0065 (LATIN SMALL LETTER E)
 - the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n - 1$ is positive or negative
 - the code units of the decimal representation of the **integer** $\text{abs}(n - 1)$ (with no leading zeroes)
10. Return the **string-concatenation** of:
- the code units of the most significant digit of the decimal representation of s
 - the code unit 0x002E (FULL STOP)
 - the code units of the remaining $k - 1$ digits of the decimal representation of s
 - the code unit 0x0065 (LATIN SMALL LETTER E)
 - the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n - 1$ is positive or negative
 - the code units of the decimal representation of the **integer** $\text{abs}(n - 1)$ (with no leading zeroes)

NOTE 1

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

- If x is any **Number value** other than $-0_{\mathbb{F}}$, then $\text{ToNumber}(\text{ToString}(x))$ 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:

5. Otherwise, let n , k , and s be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, $s \times 10^{n-k}$ is $\mathbb{R}(x)$, 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 $\mathbb{R}(x)$. If there are two such possible values of s , choose the one that is even. Note that k is the number of digits in the decimal representation of s and that s is not divisible by 10.

NOTE 3

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

Gay, David M. Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis, Manuscript 90-10. AT&T Bell Laboratories (Murray Hill, New Jersey). 30 November 1990. Available as

<http://ampl.com/REFS/abstracts.html#rounding>. Associated code available as

<http://netlib.sandia.gov/fp/dtoa.c> and as

http://netlib.sandia.gov/fp/g_fmt.c and may also be found at the various **netlib** mirror sites.

6.1.6.2 The BigInt Type

The BigInt type represents an [integer](#) value. The value may be any size and is not limited to a particular bit-width. Generally, where not otherwise noted, operations are designed to return exact mathematically-based answers. For binary operations, BigInts act as two's complement binary strings, with negative numbers treated as having bits set infinitely to the left.

The BigInt::unit value is $1_{\mathbb{Z}}$.

6.1.6.2.1 BigInt::unaryMinus (*x*)

The abstract operation BigInt::unaryMinus takes argument *x* (a BigInt). It performs the following steps when called:

1. If *x* is $0_{\mathbb{Z}}$, return $0_{\mathbb{Z}}$.
2. Return the BigInt value that represents the negation of $\mathbb{R}(x)$.

6.1.6.2.2 BigInt::bitwiseNOT (*x*)

The abstract operation BigInt::bitwiseNOT takes argument *x* (a BigInt). It returns the one's complement of *x*; that is, $-x - 1_{\mathbb{Z}}$.

6.1.6.2.3 BigInt::exponentiate (*base*, *exponent*)

The abstract operation BigInt::exponentiate takes arguments *base* (a BigInt) and *exponent* (a BigInt). It performs the following steps when called:

1. If *exponent* $< 0_{\mathbb{Z}}$, throw a **RangeError** exception.
2. If *base* is $0_{\mathbb{Z}}$ and *exponent* is $0_{\mathbb{Z}}$, return $1_{\mathbb{Z}}$.
3. Return the BigInt value that represents $\mathbb{R}(base)$ raised to the power $\mathbb{R}(exponent)$.

6.1.6.2.4 BigInt::multiply (*x*, *y*)

The abstract operation BigInt::multiply takes arguments *x* (a BigInt) and *y* (a BigInt). It returns the BigInt value that represents the result of multiplying *x* and *y*.

NOTE

Even if the result has a much larger bit width than the input, the exact mathematical answer is given.

6.1.6.2.5 `BigInt::divide (x, y)`

The abstract operation `BigInt::divide` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. If y is $0_{\mathbb{Z}}$, throw a **RangeError** exception.
2. Let *quotient* be $\mathbb{R}(x) / \mathbb{R}(y)$.
3. Return the `BigInt` value that represents *quotient* rounded towards 0 to the next **integer** value.

6.1.6.2.6 `BigInt::remainder (n, d)`

The abstract operation `BigInt::remainder` takes arguments n (a `BigInt`) and d (a `BigInt`). It performs the following steps when called:

1. If d is $0_{\mathbb{Z}}$, throw a **RangeError** exception.
2. If n is $0_{\mathbb{Z}}$, return $0_{\mathbb{Z}}$.
3. Let r be the `BigInt` defined by the mathematical relation $r = n - (d \times q)$ where q is a `BigInt` 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 .
4. Return r .

NOTE The sign of the result equals the sign of the dividend.

6.1.6.2.7 `BigInt::add (x, y)`

The abstract operation `BigInt::add` takes arguments x (a `BigInt`) and y (a `BigInt`). It returns the `BigInt` value that represents the sum of x and y .

6.1.6.2.8 `BigInt::subtract (x, y)`

The abstract operation `BigInt::subtract` takes arguments x (a `BigInt`) and y (a `BigInt`). It returns the `BigInt` value that represents the difference x minus y .

6.1.6.2.9 `BigInt::leftShift (x, y)`

The abstract operation `BigInt::leftShift` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. If $y < 0_{\mathbb{Z}}$, then
 - a. Return the `BigInt` value that represents $\mathbb{R}(x) / 2^{-y}$, rounding down to the nearest **integer**, including for negative numbers.
2. Return the `BigInt` value that represents $\mathbb{R}(x) \times 2^y$.

NOTE Semantics here should be equivalent to a bitwise shift, treating the `BigInt` as an infinite length string of binary two's complement digits.

6.1.6.2.10 `BigInt::signedRightShift (x, y)`

The abstract operation `BigInt::signedRightShift` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following

steps when called:

1. Return `BigInt::leftShift(x, -y)`.

6.1.6.2.11 `BigInt::unsignedRightShift (x, y)`

The abstract operation `BigInt::unsignedRightShift` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. Throw a `TypeError` exception.

6.1.6.2.12 `BigInt::lessThan (x, y)`

The abstract operation `BigInt::lessThan` takes arguments x (a `BigInt`) and y (a `BigInt`). It returns `true` if $\mathbb{R}(x) < \mathbb{R}(y)$ and `false` otherwise.

6.1.6.2.13 `BigInt::equal (x, y)`

The abstract operation `BigInt::equal` takes arguments x (a `BigInt`) and y (a `BigInt`). It returns `true` if $\mathbb{R}(x) = \mathbb{R}(y)$ and `false` otherwise.

6.1.6.2.14 `BigInt::sameValue (x, y)`

The abstract operation `BigInt::sameValue` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. Return `BigInt::equal(x, y)`.

6.1.6.2.15 `BigInt::sameValueZero (x, y)`

The abstract operation `BigInt::sameValueZero` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. Return `BigInt::equal(x, y)`.

6.1.6.2.16 `BinaryAnd (x, y)`

The abstract operation `BinaryAnd` takes arguments x and y . It performs the following steps when called:

1. `Assert`: x is 0 or 1.
2. `Assert`: y is 0 or 1.
3. If x is 1 and y is 1, return 1.
4. Else, return 0.

6.1.6.2.17 `BinaryOr (x, y)`

The abstract operation `BinaryOr` takes arguments x and y . It performs the following steps when called:

1. `Assert`: x is 0 or 1.
2. `Assert`: y is 0 or 1.
3. If x is 1 or y is 1, return 1.
4. Else, return 0.

6.1.6.2.18 BinaryXor (x, y)

The abstract operation BinaryXor takes arguments x and y . It performs the following steps when called:

1. **Assert:** x is 0 or 1.
2. **Assert:** y is 0 or 1.
3. If x is 1 and y is 0, return 1.
4. Else if x is 0 and y is 1, return 1.
5. Else, return 0.

6.1.6.2.19 BigIntBitwiseOp (op, x, y)

The abstract operation BigIntBitwiseOp takes arguments op (a sequence of Unicode code points), x (a BigInt), and y (a BigInt). It performs the following steps when called:

1. **Assert:** op is **&**, **^**, or **|**.
2. Set x to $\mathbb{R}(x)$.
3. Set y to $\mathbb{R}(y)$.
4. Let $result$ be 0.
5. Let $shift$ be 0.
6. Repeat, until $(x = 0$ or $x = -1)$ and $(y = 0$ or $y = -1)$,
 - a. Let $xDigit$ be x modulo 2.
 - b. Let $yDigit$ be y modulo 2.
 - c. If op is **&**, set $result$ to $result + 2^{shift} \times \text{BinaryAnd}(xDigit, yDigit)$.
 - d. Else if op is **|**, set $result$ to $result + 2^{shift} \times \text{BinaryOr}(xDigit, yDigit)$.
 - e. Else,
 - i. **Assert:** op is **^**.
 - ii. Set $result$ to $result + 2^{shift} \times \text{BinaryXor}(xDigit, yDigit)$.
 - f. Set $shift$ to $shift + 1$.
 - g. Set x to $(x - xDigit) / 2$.
 - h. Set y to $(y - yDigit) / 2$.
7. If op is **&**, let tmp be $\text{BinaryAnd}(x$ modulo 2, y modulo 2).
8. Else if op is **|**, let tmp be $\text{BinaryOr}(x$ modulo 2, y modulo 2).
9. Else,
 - a. **Assert:** op is **^**.
 - b. Let tmp be $\text{BinaryXor}(x$ modulo 2, y modulo 2).
10. If $tmp \neq 0$, then
 - a. Set $result$ to $result - 2^{shift}$.
 - b. NOTE: This extends the sign.
11. Return the BigInt value for $result$.

6.1.6.2.20 BigInt::bitwiseAND (x, y)

The abstract operation BigInt::bitwiseAND takes arguments x (a BigInt) and y (a BigInt). It performs the following steps when called:

1. Return `BigIntBitwiseOp(&, x, y)`.

6.1.6.2.21 BigInt::bitwiseXOR (x, y)

The abstract operation `BigInt::bitwiseXOR` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. Return `BigIntBitwiseOp(\wedge , x , y)`.

6.1.6.2.22 `BigInt::bitwiseOR` (x , y)

The abstract operation `BigInt::bitwiseOR` takes arguments x (a `BigInt`) and y (a `BigInt`). It performs the following steps when called:

1. Return `BigIntBitwiseOp(\vee , x , y)`.

6.1.6.2.23 `BigInt::toString` (x)

The abstract operation `BigInt::toString` takes argument x (a `BigInt`). It converts x to `String` format. It performs the following steps when called:

1. If $x < 0_{\mathbb{Z}}$, return the [string-concatenation](#) of the `String` `"-"` and `! BigInt::toString(-x)`.
2. Return the `String` value consisting of the code units of the digits of the decimal representation of x .

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 property 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. A *property name* is a property key that is a `String` value.

An *integer index* is a `String`-valued property key that is a canonical numeric `String` (see [7.1.21](#)) and whose numeric value is either $+0_{\mathbb{F}}$ or a positive [integral Number](#) $\leq \mathbb{F}(2^{53} - 1)$. An *array index* is an [integer index](#) whose numeric value i is in the range $+0_{\mathbb{F}} \leq i < \mathbb{F}(2^{32} - 1)$.

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 property 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. Please see [6.1.7.2](#) for definitions of the multiple forms of objects.

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 3](#).

Table 3: Attributes of a Data Property

Attribute Name	Value Domain	Description
[[Value]]	Any ECMAScript language type	The value retrieved by a get access of the property.
[[Writable]]	Boolean	If false , attempts by ECMAScript code to change the property's [[Value]] attribute using [[Set]] will not succeed.
[[Enumerable]]	Boolean	If true , the property will be enumerated by a for-in enumeration (see 14.7.5). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If false , attempts to delete the property, change the property to be an accessor property , or change its attributes (other than [[Value]], or changing [[Writable]] to false) will fail.

An [accessor property](#) associates a key value with the attributes listed in [Table 4](#).

Table 4: Attributes of an Accessor Property

Attribute Name	Value Domain	Description
[[Get]]	Object Undefined	If the value is an Object it must be a function object . The function's [[Call]] internal method (Table 7) is called with an empty arguments list to retrieve the property value each time a get access of the property is performed.
[[Set]]	Object Undefined	If the value is an Object it must be a function object . The function's [[Call]] internal method (Table 7) 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.
[[Enumerable]]	Boolean	If true , the property is to be enumerated by a for-in enumeration (see 14.7.5). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If false , attempts to delete the property, change the property to be a data property , or change its attributes will fail.

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in [Table 5](#) is used.

Table 5: Default Attribute Values

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

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. That actual object upon which an internal method is invoked is the “target” of the invocation. If, at runtime, the implementation of an algorithm attempts to use an internal method of an object that the object does not support, a **TypeError** exception is thrown.

Internal slots correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms. Internal slots are not object properties and they are not inherited. Depending upon the specific internal slot specification, such state may consist of values of any [ECMAScript language type](#) or of specific ECMAScript specification type values. Unless explicitly specified otherwise, internal slots are allocated as part of the process of creating an object and may not be dynamically added to an object. Unless specified otherwise, the initial value of an internal slot is the value **undefined**. Various algorithms within this specification create objects that have internal slots. However, the ECMAScript language provides no direct way to associate internal slots with an object.

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

[Table 6](#) 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.

An *ordinary object* is an object that satisfies all of the following criteria:

- For the internal methods listed in [Table 6](#), the object uses those defined in [10.1](#).
- If the object has a `[[Call]]` internal method, it uses the one defined in [10.2.1](#).
- If the object has a `[[Construct]]` internal method, it uses the one defined in [10.2.2](#).

An *exotic object* is an object that is not an [ordinary object](#).

This specification recognizes different kinds of exotic objects by those objects' internal methods. An object that is

behaviourally equivalent to a particular kind of [exotic object](#) (such as an [Array exotic object](#) or a [bound function exotic object](#)), but does not have the same collection of internal methods specified for that kind, is not recognized as that kind of [exotic object](#).

The “Signature” column of [Table 6](#) 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](#).

In addition to its parameters, an internal method always has access to the object that is the target of the method invocation.

An internal method implicitly returns a [Completion Record](#), either a normal completion that wraps a value of the return type shown in its invocation pattern, or a throw completion.

Table 6: Essential Internal Methods

Internal Method	Signature	Description
[[GetPrototypeOf]]	() → Object Null	Determine the object that provides inherited properties for this object. A null value indicates that there are no inherited properties.
[[SetPrototypeOf]]	(Object Null) → Boolean	Associate this object with 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.
[[IsExtensible]]	() → Boolean	Determine whether it is permitted to add additional properties to this object.
[[PreventExtensions]]	() → Boolean	Control whether new properties may be added to this object. Returns true if the operation was successful or false if the operation was unsuccessful.
[[GetOwnProperty]]	(<i>propertyKey</i>) → Undefined Property Descriptor	Return a Property Descriptor for the own property of this object whose key is <i>propertyKey</i> , or undefined if no such property exists.
[[DefineOwnProperty]]	(<i>propertyKey</i> , <i>PropertyDescriptor</i>) → Boolean	Create or alter the own property, whose key is <i>propertyKey</i> , to have the state described by <i>PropertyDescriptor</i> . Return true if that property was successfully created / updated or false if the property could not be created or updated.
[[HasProperty]]	(<i>propertyKey</i>) → Boolean	Return a Boolean value indicating whether this object already has either an own or inherited property whose key is <i>propertyKey</i> .
[[Get]]	(<i>propertyKey</i> , <i>Receiver</i>) → any	Return the value of the property whose key is <i>propertyKey</i> from this object. If any ECMAScript code must be executed to retrieve the property value, <i>Receiver</i> is used as the this value when evaluating the code.
[[Set]]	(<i>propertyKey</i> , <i>value</i> , <i>Receiver</i>) → Boolean	Set the value of the property whose key is <i>propertyKey</i> to <i>value</i> . If any ECMAScript code must be executed to set the property value, <i>Receiver</i> is used as the this value when evaluating the code. Returns true if the property value was set or false if it could not be set.
[[Delete]]	(<i>propertyKey</i>) → Boolean	Remove the own property whose key is <i>propertyKey</i> from this object. Return false if the property was not deleted and is still present. Return true if the property was deleted or is not present.
[[OwnPropertyKeys]]	() → List of propertyKey	Return a List whose elements are all of the own property keys for the object.

Table 7 summarizes additional essential internal methods that are supported by objects that may be called as functions. A *function object* is an object that supports the [[Call]] internal method. A *constructor* is an object that supports the [[Construct]] internal method. Every object that supports [[Construct]] must support [[Call]]; that is,

every **constructor** must be a **function object**. Therefore, a **constructor** may also be referred to as a *constructor function* or *constructor function object*.

Table 7: Additional Essential Internal Methods of Function Objects

Internal Method	Signature	Description
[[Call]]	(<i>any</i> , a List of <i>any</i>) → <i>any</i>	Executes code associated with this object. Invoked via a function call expression. The arguments to the internal method are a this value and a List whose elements are the arguments passed to the function by a call expression. Objects that implement this internal method are <i>callable</i> .
[[Construct]]	(a List of <i>any</i> , Object) → Object	Creates an object. Invoked via the new operator or a super call. The first argument to the internal method is a List whose elements are the arguments of the constructor invocation or the super call. The second argument is the object to which the new operator was initially applied. Objects that implement this internal method are called <i>constructors</i> . A function object is not necessarily a constructor and such non- constructor function objects do not have a [[Construct]] internal method.

The semantics of the essential internal methods for ordinary objects and standard exotic objects are specified in clause 10. If any specified use of an internal method of an **exotic object** 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.

An implementation must not allow these invariants to be circumvented in any manner such as by providing alternative interfaces that implement the functionality of the essential internal methods without enforcing their invariants.

Definitions:

- The *target* of an internal method is the object upon which the internal method is called.
- 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 the **SameValue** algorithm.

Return value:

The value returned by any internal method must be a **Completion Record** with either:

- `[[Type]]` = normal, `[[Target]]` = empty, and `[[Value]]` = a value of the "normal return type" shown below for that internal method, or
- `[[Type]]` = throw, `[[Target]]` = empty, and `[[Value]]` = any [ECMAScript language value](#).

NOTE 1 An internal method must not return a completion with `[[Type]]` = continue, break, or return.

`[[GetPrototypeOf]]` ()

- The normal return type is 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*.

NOTE 2 An object's prototype chain should have finite length (that is, starting from any object, recursively applying the `[[GetPrototypeOf]]` internal method to its result should eventually lead to the value **null**). However, this requirement is not enforceable as an object level invariant if the prototype chain includes any exotic objects that do not use the [ordinary object](#) definition of `[[GetPrototypeOf]]`. Such a circular prototype chain may result in infinite loops when accessing object properties.

`[[SetPrototypeOf]]` (*V*)

- The normal return type is Boolean.
- If target is non-extensible, `[[SetPrototypeOf]]` must return **false**, unless *V* is the [SameValue](#) as the target's observed `[[GetPrototypeOf]]` value.

`[[IsExtensible]]` ()

- The normal return type is Boolean.
- If `[[IsExtensible]]` returns **false**, all future calls to `[[IsExtensible]]` on the target must return **false**.

`[[PreventExtensions]]` ()

- The normal return type is 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 normal return type is either [Property Descriptor](#) or Undefined.
- If the Type of the return value is [Property Descriptor](#), the return value must be a fully populated [Property Descriptor](#).
- If *P* is described as a non-configurable, non-writable own [data property](#), all future calls to `[[GetOwnProperty]]` (*P*) must return [Property Descriptor](#) whose `[[Value]]` is [SameValue](#) as *P*'s `[[Value]]` attribute.
- If *P*'s attributes other than `[[Writable]]` may change over time or if the property might be deleted, 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**).

NOTE 3

As a consequence of the third invariant, if a property is described as a [data property](#) and it may return different values over time, then either or both of the `[[Writable]]` and `[[Configurable]]` attributes must be **true** even if no mechanism to change the value is exposed via the other essential internal methods.

`[[DefineOwnProperty]] (P, Desc)`

- The normal return type is 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 writable [data property](#). A non-configurable writable [data property](#) can be changed into a non-configurable non-writable [data property](#).
 2. All attributes of *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 normal return type is Boolean.
- If *P* was previously observed as a non-configurable own data or [accessor property](#) of the target, `[[HasProperty]]` must return **true**.

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

- The normal return type is any [ECMAScript language type](#).
- 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](#) as *V*.
- 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 normal return type is 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 normal return type is Boolean.
- If *P* was previously observed as a non-configurable own data or [accessor property](#) of the target, `[[Delete]]` must return **false**.

`[[OwnPropertyKeys]] ()`

- The normal return type is [List](#).
- The returned [List](#) must not contain any duplicate entries.
- 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 target is non-extensible, the returned [List](#) must contain only the keys of all own properties of the target that are observable using `[[GetOwnProperty]]`.

`[[Call]]` ()

- The normal return type is any [ECMAScript language type](#).

`[[Construct]]` ()

- The normal return type is `Object`.
- The target must also have a `[[Call]]` internal method.

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. A reference such as `%name.a.b%` means, as if the "b" property of the "a" property of the intrinsic object `%name%` was accessed prior to any ECMAScript code being evaluated. Determination of the current [realm](#) and its intrinsics is described in [9.3](#). The well-known intrinsics are listed in [Table 8](#).

Table 8: Well-Known Intrinsic Objects

Intrinsic Name	Global Name	ECMAScript Language Association
<code>%AggregateError%</code>	AggregateError	The AggregateError constructor (20.5.7.1)
<code>%Array%</code>	Array	The Array constructor (23.1.1)
<code>%ArrayBuffer%</code>	ArrayBuffer	The ArrayBuffer constructor (25.1.3)
<code>%ArrayIteratorPrototype%</code>		The prototype of Array iterator objects (23.1.5)
<code>%AsyncFromSyncIteratorPrototype%</code>		The prototype of async-from-sync iterator objects (27.1.4)
<code>%AsyncFunction%</code>		The constructor of async function objects (27.7.1)
<code>%AsyncGeneratorFunction%</code>		The constructor of async iterator objects (27.4.1)
<code>%AsyncIteratorPrototype%</code>		An object that all standard built-in async iterator objects indirectly inherit from
<code>%Atomics%</code>	Atomics	The Atomics object (25.4)
<code>%BigInt%</code>	BigInt	The BigInt constructor (21.2.1)
<code>%BigInt64Array%</code>	BigInt64Array	The BigInt64Array constructor (23.2)

%BigInt64Array%	BigInt64Array	The BigInt64Array constructor (23.2)
%Boolean%	Boolean	The Boolean constructor (20.3.1)
%DataView%	DataView	The DataView constructor (25.3.2)
%Date%	Date	The Date constructor (21.4.2)
%decodeURI%	decodeURI	The decodeURI function (19.2.6.2)
%decodeURIComponent%	decodeURIComponent	The decodeURIComponent function (19.2.6.3)
%encodeURIComponent%	encodeURIComponent	The encodeURIComponent function (19.2.6.4)
%encodeURIComponent%	encodeURIComponent	The encodeURIComponent function (19.2.6.5)
%Error%	Error	The Error constructor (20.5.1)
%eval%	eval	The eval function (19.2.1)
%EvalError%	EvalError	The EvalError constructor (20.5.5.1)
%FinalizationRegistry%	FinalizationRegistry	The FinalizationRegistry constructor (26.2.1)
%Float32Array%	Float32Array	The Float32Array constructor (23.2)
%Float64Array%	Float64Array	The Float64Array constructor (23.2)
%ForInIteratorPrototype%		The prototype of For-In iterator objects (14.7.5.10)
%Function%	Function	The Function constructor (20.2.1)
%GeneratorFunction%		The constructor of generator objects (27.3.1)
%Int8Array%	Int8Array	The Int8Array constructor (23.2)
%Int16Array%	Int16Array	The Int16Array constructor (23.2)
%Int32Array%	Int32Array	The Int32Array constructor (23.2)
%isFinite%	isFinite	The isFinite function (19.2.2)
%isNaN%	isNaN	The isNaN function (19.2.3)
%IteratorPrototype%		An object that all standard built-in iterator objects indirectly inherit from
%JSON%	JSON	The JSON object (25.5)
%Map%	Map	The Map constructor (24.1.1)
%MapIteratorPrototype%		The prototype of Map iterator objects (24.1.5)

<code>%Math%</code>	Math	The Math object (21.3)
<code>%Number%</code>	Number	The Number constructor (21.1.1)
<code>%Object%</code>	Object	The Object constructor (20.1.1)
<code>%parseFloat%</code>	parseFloat	The parseFloat function (19.2.4)
<code>%parseInt%</code>	parseInt	The parseInt function (19.2.5)
<code>%Promise%</code>	Promise	The Promise constructor (27.2.3)
<code>%Proxy%</code>	Proxy	The Proxy constructor (28.2.1)
<code>%RangeError%</code>	RangeError	The RangeError constructor (20.5.5.2)
<code>%ReferenceError%</code>	ReferenceError	The ReferenceError constructor (20.5.5.3)
<code>%Reflect%</code>	Reflect	The Reflect object (28.1)
<code>%RegExp%</code>	RegExp	The RegExp constructor (22.2.3)
<code>%RegExpStringIteratorPrototype%</code>		The prototype of RegExp String Iterator objects (22.2.7)
<code>%Set%</code>	Set	The Set constructor (24.2.1)
<code>%SetIteratorPrototype%</code>		The prototype of Set iterator objects (24.2.5)
<code>%SharedArrayBuffer%</code>	SharedArrayBuffer	The SharedArrayBuffer constructor (25.2.2)
<code>%String%</code>	String	The String constructor (22.1.1)
<code>%StringIteratorPrototype%</code>		The prototype of String iterator objects (22.1.5)
<code>%Symbol%</code>	Symbol	The Symbol constructor (20.4.1)
<code>%SyntaxError%</code>	SyntaxError	The SyntaxError constructor (20.5.5.4)
<code>%ThrowTypeError%</code>		A function object that unconditionally throws a new instance of <code>%TypeError%</code>
<code>%TypedArray%</code>		The super class of all typed Array constructors (23.2.1)
<code>%TypeError%</code>	TypeError	The TypeError constructor (20.5.5.5)
<code>%Uint8Array%</code>	Uint8Array	The Uint8Array constructor (23.2)
<code>%Uint8ClampedArray%</code>	Uint8ClampedArray	The Uint8ClampedArray constructor (23.2)
<code>%Uint16Array%</code>	Uint16Array	The Uint16Array constructor (23.2)
<code>%Uint32Array%</code>	Uint32Array	The Uint32Array constructor (23.2)
<code>%URIError%</code>	URIError	The URIError constructor (20.5.5.6)

%WeakMap%	WeakMap	The WeakMap constructor (24.3.1)
%WeakRef%	WeakRef	The WeakRef constructor (26.1.1)
%WeakSet%	WeakSet	The WeakSet constructor (24.4.1)

NOTE Additional entries in [Table 82](#).

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 include Reference, [List](#), [Completion](#), [Property Descriptor](#), [Environment Record](#), [Abstract Closure](#), 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 Types

The *List* type is used to explain the evaluation of argument lists (see [13.3.8](#)) 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*.

When an algorithm iterates over the elements of a List without specifying an order, the order used is the order of the elements in the List.

For notational convenience within this specification, a literal syntax can be used to express a new List value. For example, « 1, 2 » defines a List value that has two elements each of which is initialized to a specific value. A new empty List can be expressed as « ».

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

For notational convenience within this specification, an object literal-like syntax can be used to express a Record value. For example, { [[Field1]]: 42, [[Field2]]: false, [[Field3]]: empty } defines a Record value that has three fields, each of which is initialized to a specific value. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

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

Schema for commonly used Record field combinations may be named, and that name may be used as a prefix to a

literal Record value to identify the specific kind of aggregations that is being described. For example: PropertyDescriptor { [[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true** }.

6.2.2 The Set and Relation Specification Types

The *Set* type is used to explain a collection of unordered elements for use in the [memory model](#). Values of the Set type are simple collections of elements, where no element appears more than once. Elements may be added to and removed from Sets. Sets may be unioned, intersected, or subtracted from each other.

The *Relation* type is used to explain constraints on Sets. Values of the Relation type are Sets of ordered pairs of values from its value domain. For example, a Relation on events is a set of ordered pairs of events. For a Relation R and two values a and b in the value domain of R , $a R b$ is shorthand for saying the ordered pair (a, b) is a member of R . A Relation is least with respect to some conditions when it is the smallest Relation that satisfies those conditions.

A *strict partial order* is a Relation value R that satisfies the following.

- For all a, b , and c in R 's domain:
 - It is not the case that $a R a$, and
 - If $a R b$ and $b R c$, then $a R c$.

NOTE 1 The two properties above are called irreflexivity and transitivity, respectively.

A *strict total order* is a Relation value R that satisfies the following.

- For all a, b , and c in R 's domain:
 - a is identical to b or $a R b$ or $b R a$, and
 - It is not the case that $a R a$, and
 - If $a R b$ and $b R c$, then $a R c$.

NOTE 2 The three properties above are called totality, irreflexivity, and transitivity, respectively.

6.2.3 The Completion Record Specification Type

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

Values of the Completion type are [Record](#) values whose fields are defined by [Table 9](#). Such values are referred to as *Completion Records*.

Table 9: [Completion Record](#) Fields

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

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

6.2.3.1 Await

Algorithm steps that say

1. Let *completion* be `Await(value)`.

mean the same thing as:

1. Let *asyncContext* be the `running execution context`.
2. Let *promise* be `? PromiseResolve(%Promise%, value)`.
3. Let *stepsFulfilled* be the algorithm steps defined in `Await Fulfilled Functions`.
4. Let *lengthFulfilled* be the number of non-optional parameters of the function definition in `Await Fulfilled Functions`.
5. Let *onFulfilled* be `! CreateBuiltinFunction(stepsFulfilled, lengthFulfilled, "", « [[AsyncContext]] »)`.
6. Set *onFulfilled*.`[[AsyncContext]]` to *asyncContext*.
7. Let *stepsRejected* be the algorithm steps defined in `Await Rejected Functions`.
8. Let *lengthRejected* be the number of non-optional parameters of the function definition in `Await Rejected Functions`.
9. Let *onRejected* be `! CreateBuiltinFunction(stepsRejected, lengthRejected, "", « [[AsyncContext]] »)`.
10. Set *onRejected*.`[[AsyncContext]]` to *asyncContext*.
11. Perform `! PerformPromiseThen(promise, onFulfilled, onRejected)`.
12. Remove *asyncContext* 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`.
13. Set the code evaluation state of *asyncContext* such that when evaluation is resumed with a `Completion completion`, the following steps of the algorithm that invoked `Await` will be performed, with *completion* available.
14. Return.
15. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of *asyncContext*.

where all aliases in the above steps, with the exception of *completion*, are ephemeral and visible only in the steps pertaining to `Await`.

NOTE

`Await` can be combined with the `?` and `!` prefixes, so that for example

1. Let *result* be `? Await(value)`.

means the same thing as:

1. Let *result* be `Await(value)`.
2. `ReturnIfAbrupt(result)`.

6.2.3.1.1 Await Fulfilled Functions

An `Await` fulfilled function is an anonymous built-in function that is used as part of the `Await` specification device to deliver the promise fulfillment value to the caller as a normal completion. Each `Await` fulfilled function has an `[[AsyncContext]]` internal slot.

When an `Await` fulfilled function is called with argument *value*, the following steps are taken:

1. Let *F* be the `active function object`.
2. Let *asyncContext* be *F*.[[AsyncContext]].
3. Let *prevContext* be the `running execution context`.
4. Suspend *prevContext*.
5. Push *asyncContext* onto the `execution context stack`; *asyncContext* is now the `running execution context`.
6. Resume the suspended evaluation of *asyncContext* using `NormalCompletion(value)` as the result of the operation that suspended it.
7. **Assert:** When we reach this step, *asyncContext* has already been removed from the `execution context stack` and *prevContext* is the currently `running execution context`.
8. Return **undefined**.

The "length" property of an `Await` fulfilled function is 1_F.

6.2.3.1.2 `Await` Rejected Functions

An `Await` rejected function is an anonymous built-in function that is used as part of the `Await` specification device to deliver the promise rejection reason to the caller as an abrupt throw completion. Each `Await` rejected function has an [[AsyncContext]] internal slot.

When an `Await` rejected function is called with argument *reason*, the following steps are taken:

1. Let *F* be the `active function object`.
2. Let *asyncContext* be *F*.[[AsyncContext]].
3. Let *prevContext* be the `running execution context`.
4. Suspend *prevContext*.
5. Push *asyncContext* onto the `execution context stack`; *asyncContext* is now the `running execution context`.
6. Resume the suspended evaluation of *asyncContext* using `ThrowCompletion(reason)` as the result of the operation that suspended it.
7. **Assert:** When we reach this step, *asyncContext* has already been removed from the `execution context stack` and *prevContext* is the currently `running execution context`.
8. Return **undefined**.

The "length" property of an `Await` rejected function is 1_F.

6.2.3.2 `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 }.

6.2.3.3 `ThrowCompletion`

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

1. Return `ThrowCompletion(argument)`.

Is a shorthand that is defined as follows:

1. Return **Completion** { [[Type]]: throw, [[Value]]: *argument*, [[Target]]: empty }.

6.2.3.4 UpdateEmpty (*completionRecord*, *value*)

The abstract operation UpdateEmpty takes arguments *completionRecord* and *value*. It performs the following steps when called:

1. **Assert**: If *completionRecord*.[[Type]] is either return or throw, then *completionRecord*.[[Value]] is not empty.
2. If *completionRecord*.[[Value]] is not empty, return **Completion**(*completionRecord*).
3. Return **Completion** { [[Type]]: *completionRecord*.[[Type]], [[Value]]: *value*, [[Target]]: *completionRecord*.[[Target]] }.

6.2.4 The Reference Record Specification Type

The *Reference Record* 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 Record.

A Reference Record is a resolved name or property binding; its fields are defined by Table 10.

Table 10: Reference Record Fields

Field Name	Value	Meaning
[[Base]]	One of: <ul style="list-style-type: none"> • any ECMAScript language value except undefined or null, • an Environment Record, or • unresolvable. 	The value or Environment Record which holds the binding. A [[Base]] of unresolvable indicates that the binding could not be resolved.
[[ReferencedName]]	String or Symbol	The name of the binding. Always a String if [[Base]] value is an Environment Record .
[[Strict]]	Boolean	true if the Reference Record originated in strict mode code , false otherwise.
[[ThisValue]]	any ECMAScript language value or empty	If not empty , the Reference Record represents a property binding that was expressed using the super keyword; it is called a <i>Super Reference Record</i> and its [[Base]] value will never be an Environment Record . In that case, the [[ThisValue]] field holds the this value at the time the Reference Record was created.

The following [abstract operations](#) are used in this specification to operate upon References:

6.2.4.1 IsPropertyReference (*V*)

The abstract operation IsPropertyReference takes argument *V*. It performs the following steps when called:

1. **Assert:** *V* is a [Reference Record](#).
2. If *V*.[[Base]] is **unresolvable**, return **false**.
3. If [Type](#)(*V*.[[Base]]) is Boolean, String, Symbol, BigInt, Number, or Object, return **true**; otherwise return **false**.

6.2.4.2 IsUnresolvableReference (*V*)

The abstract operation IsUnresolvableReference takes argument *V*. It performs the following steps when called:

1. **Assert:** *V* is a [Reference Record](#).
2. If *V*.[[Base]] is **unresolvable**, return **true**; otherwise return **false**.

6.2.4.3 IsSuperReference (*V*)

The abstract operation IsSuperReference takes argument *V*. It performs the following steps when called:

1. **Assert:** *V* is a [Reference Record](#).
2. If *V*.[[ThisValue]] is not **empty**, return **true**; otherwise return **false**.

6.2.4.4 GetValue (*V*)

The abstract operation GetValue takes argument *V*. It performs the following steps when called:

1. [ReturnIfAbrupt](#)(*V*).
2. If *V* is not a [Reference Record](#), return *V*.
3. If [IsUnresolvableReference](#)(*V*) is **true**, throw a **ReferenceError** exception.
4. If [IsPropertyReference](#)(*V*) is **true**, then
 - a. Let *baseObj* be ! [ToObject](#)(*V*.[[Base]]).
 - b. Return ? *baseObj*.[[Get]](*V*.[[ReferencedName]], [GetThisValue](#)(*V*)).
5. Else,
 - a. Let *base* be *V*.[[Base]].
 - b. **Assert:** *base* is an [Environment Record](#).
 - c. Return ? *base*.GetBindingValue(*V*.[[ReferencedName]], *V*.[[Strict]]) (see 9.1).

NOTE

The object that may be created in step 4.a 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.

6.2.4.5 PutValue (*V*, *W*)

The abstract operation PutValue takes arguments *V* and *W*. It performs the following steps when called:

1. [ReturnIfAbrupt](#)(*V*).
2. [ReturnIfAbrupt](#)(*W*).

3. If V is not a [Reference Record](#), throw a **ReferenceError** exception.
4. If `IsUnresolvableReference(V)` is **true**, then
 - a. If V .[[Strict]] is **true**, throw a **ReferenceError** exception.
 - b. Let $globalObj$ be `GetGlobalObject()`.
 - c. Return ? `Set(globalObj, V .[[ReferencedName]], W , false)`.
5. If `IsPropertyReference(V)` is **true**, then
 - a. Let $baseObj$ be ! `ToObject(V .[[Base]])`.
 - b. Let $succeeded$ be ? $baseObj$.[[Set]](V .[[ReferencedName]], W , `GetThisValue(V)`).
 - c. If $succeeded$ is **false** and V .[[Strict]] is **true**, throw a **TypeError** exception.
 - d. Return.
6. Else,
 - a. Let $base$ be V .[[Base]].
 - b. **Assert**: $base$ is an [Environment Record](#).
 - c. Return ? $base$.SetMutableBinding(V .[[ReferencedName]], W , V .[[Strict]]) (see 9.1).

NOTE

The object that may be created in step 5.a is not accessible outside of the above abstract operation and the [ordinary object](#) [[Set]] internal method. An implementation might choose to avoid the actual creation of that object.

6.2.4.6 GetThisValue (V)

The abstract operation `GetThisValue` takes argument V . It performs the following steps when called:

1. **Assert**: `IsPropertyReference(V)` is **true**.
2. If `IsSuperReference(V)` is **true**, return V .[[ThisValue]]; otherwise return V .[[Base]].

6.2.4.7 InitializeReferencedBinding (V , W)

The abstract operation `InitializeReferencedBinding` takes arguments V and W . It performs the following steps when called:

1. `ReturnIfAbrupt(V)`.
2. `ReturnIfAbrupt(W)`.
3. **Assert**: V is a [Reference Record](#).
4. **Assert**: `IsUnresolvableReference(V)` is **false**.
5. Let $base$ be V .[[Base]].
6. **Assert**: $base$ is an [Environment Record](#).
7. Return $base$.InitializeBinding(V .[[ReferencedName]], W).

6.2.5 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 3](#) or [Table 4](#).

The following [abstract operations](#) are used in this specification to operate upon Property Descriptor values:

6.2.5.1 IsAccessorDescriptor (*Desc*)

The abstract operation IsAccessorDescriptor takes argument *Desc* (a [Property Descriptor](#) or **undefined**). It performs the following steps when called:

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

6.2.5.2 IsDataDescriptor (*Desc*)

The abstract operation IsDataDescriptor takes argument *Desc* (a [Property Descriptor](#) or **undefined**). It performs the following steps when called:

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

6.2.5.3 IsGenericDescriptor (*Desc*)

The abstract operation IsGenericDescriptor takes argument *Desc* (a [Property Descriptor](#) or **undefined**). It performs the following steps when called:

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

6.2.5.4 FromPropertyDescriptor (*Desc*)

The abstract operation FromPropertyDescriptor takes argument *Desc* (a [Property Descriptor](#) or **undefined**). It performs the following steps when called:

1. If *Desc* is **undefined**, return **undefined**.
2. Let *obj* be `! OrdinaryObjectCreate(%Object.prototype%)`.
3. **Assert**: *obj* is an extensible [ordinary object](#) with no own properties.
4. If *Desc* has a `[[Value]]` field, then
 - a. Perform `! CreateDataPropertyOrThrow(obj, "value", Desc.[[Value]])`.
5. If *Desc* has a `[[Writable]]` field, then
 - a. Perform `! CreateDataPropertyOrThrow(obj, "writable", Desc.[[Writable]])`.
6. If *Desc* has a `[[Get]]` field, then

- a. Perform ! `CreateDataPropertyOrThrow(obj, "get", Desc.[[Get]])`.
7. If *Desc* has a `[[Set]]` field, then
 - a. Perform ! `CreateDataPropertyOrThrow(obj, "set", Desc.[[Set]])`.
8. If *Desc* has an `[[Enumerable]]` field, then
 - a. Perform ! `CreateDataPropertyOrThrow(obj, "enumerable", Desc.[[Enumerable]])`.
9. If *Desc* has a `[[Configurable]]` field, then
 - a. Perform ! `CreateDataPropertyOrThrow(obj, "configurable", Desc.[[Configurable]])`.
10. Return *obj*.

6.2.5.5 ToPropertyDescriptor (*Obj*)

The abstract operation `ToPropertyDescriptor` takes argument *Obj*. It performs the following steps when called:

1. If `Type(Obj)` is not `Object`, throw a `TypeError` exception.
2. Let *desc* be a new `Property Descriptor` that initially has no fields.
3. Let *hasEnumerable* be ? `HasProperty(Obj, "enumerable")`.
4. If *hasEnumerable* is `true`, then
 - a. Let *enumerable* be ! `ToBoolean(? Get(Obj, "enumerable"))`.
 - b. Set *desc*.`[[Enumerable]]` to *enumerable*.
5. Let *hasConfigurable* be ? `HasProperty(Obj, "configurable")`.
6. If *hasConfigurable* is `true`, then
 - a. Let *configurable* be ! `ToBoolean(? Get(Obj, "configurable"))`.
 - b. Set *desc*.`[[Configurable]]` to *configurable*.
7. Let *hasValue* be ? `HasProperty(Obj, "value")`.
8. If *hasValue* is `true`, then
 - a. Let *value* be ? `Get(Obj, "value")`.
 - b. Set *desc*.`[[Value]]` to *value*.
9. Let *hasWritable* be ? `HasProperty(Obj, "writable")`.
10. If *hasWritable* is `true`, then
 - a. Let *writable* be ! `ToBoolean(? Get(Obj, "writable"))`.
 - b. Set *desc*.`[[Writable]]` to *writable*.
11. Let *hasGet* be ? `HasProperty(Obj, "get")`.
12. If *hasGet* is `true`, then
 - a. Let *getter* be ? `Get(Obj, "get")`.
 - b. If `IsCallable(getter)` is `false` and *getter* is not `undefined`, throw a `TypeError` exception.
 - c. Set *desc*.`[[Get]]` to *getter*.
13. Let *hasSet* be ? `HasProperty(Obj, "set")`.
14. If *hasSet* is `true`, then
 - a. Let *setter* be ? `Get(Obj, "set")`.
 - b. If `IsCallable(setter)` is `false` and *setter* is not `undefined`, throw a `TypeError` exception.
 - c. Set *desc*.`[[Set]]` to *setter*.
15. If *desc*.`[[Get]]` is present or *desc*.`[[Set]]` is present, then
 - a. If *desc*.`[[Value]]` is present or *desc*.`[[Writable]]` is present, throw a `TypeError` exception.
16. Return *desc*.

6.2.5.6 CompletePropertyDescriptor (*Desc*)

The abstract operation `CompletePropertyDescriptor` takes argument *Desc* (a `Property Descriptor`). It performs the

following steps when called:

1. **Assert:** *Desc* is a [Property Descriptor](#).
2. Let *like* be the [Record](#) { `[[Value]]: undefined`, `[[Writable]]: false`, `[[Get]]: undefined`, `[[Set]]: undefined`, `[[Enumerable]]: false`, `[[Configurable]]: false` }.
3. If `IsGenericDescriptor(Desc)` is **true** or `IsDataDescriptor(Desc)` is **true**, then
 - a. If *Desc* does not have a `[[Value]]` field, set *Desc*.`[[Value]]` to *like*.`[[Value]]`.
 - b. If *Desc* does not have a `[[Writable]]` field, set *Desc*.`[[Writable]]` to *like*.`[[Writable]]`.
4. Else,
 - a. If *Desc* does not have a `[[Get]]` field, set *Desc*.`[[Get]]` to *like*.`[[Get]]`.
 - b. If *Desc* does not have a `[[Set]]` field, set *Desc*.`[[Set]]` to *like*.`[[Set]]`.
5. If *Desc* does not have an `[[Enumerable]]` field, set *Desc*.`[[Enumerable]]` to *like*.`[[Enumerable]]`.
6. If *Desc* does not have a `[[Configurable]]` field, set *Desc*.`[[Configurable]]` to *like*.`[[Configurable]]`.
7. Return *Desc*.

6.2.6 The Environment Record Specification Type

The [Environment Record](#) type is used to explain the behaviour of name resolution in nested functions and blocks. This type and the operations upon it are defined in [9.1](#).

6.2.7 The Abstract Closure Specification Type

The *Abstract Closure* specification type is used to refer to algorithm steps together with a collection of values. Abstract Closures are meta-values and are invoked using function application style such as `closure(arg1, arg2)`. Like [abstract operations](#), invocations perform the algorithm steps described by the Abstract Closure.

In algorithm steps that create an Abstract Closure, values are captured with the verb "capture" followed by a list of aliases. When an Abstract Closure is created, it captures the value that is associated with each alias at that time. In steps that specify the algorithm to be performed when an Abstract Closure is called, each captured value is referred to by the alias that was used to capture the value.

If an Abstract Closure returns a [Completion Record](#), that [Completion Record](#)'s `[[Type]]` must be either `normal` or `throw`.

Abstract Closures are created inline as part of other algorithms, shown in the following example.

1. Let *addend* be 41.
2. Let *closure* be a new [Abstract Closure](#) with parameters (*x*) that captures *addend* and performs the following steps when called:
 - a. Return $x + \textit{addend}$.
3. Let *val* be `closure(1)`.
4. **Assert:** *val* is 42.

6.2.8 Data Blocks

The *Data Block* specification type is used to describe a distinct and mutable sequence of byte-sized (8 bit) numeric values. A *byte value* is an [integer](#) value in the range 0 through 255, inclusive. 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 access the individual bytes of

a Data Block value. This notation presents a Data Block value as a 0-origin [integer](#)-indexed sequence of bytes. For example, if *db* is a 5 byte Data Block value then *db*[2] can be used to access its 3rd byte.

A data block that resides in memory that can be referenced from multiple agents concurrently is designated a *Shared Data Block*. A Shared Data Block has an identity (for the purposes of equality testing Shared Data Block values) that is *address-free*: it is tied not to the virtual addresses the block is mapped to in any process, but to the set of locations in memory that the block represents. Two data blocks are equal only if the sets of the locations they contain are equal; otherwise, they are not equal and the intersection of the sets of locations they contain is empty. Finally, Shared Data Blocks can be distinguished from Data Blocks.

The semantics of Shared Data Blocks is defined using Shared Data Block events by the [memory model](#). [Abstract operations](#) below introduce Shared Data Block events and act as the interface between evaluation semantics and the event semantics of the [memory model](#). The events form a [candidate execution](#), on which the [memory model](#) acts as a filter. Please consult the [memory model](#) for full semantics.

Shared Data Block events are modeled by Records, defined in the [memory model](#).

The following [abstract operations](#) are used in this specification to operate upon Data Block values:

6.2.8.1 CreateByteDataBlock (*size*)

The abstract operation CreateByteDataBlock takes argument *size* (an [integer](#)). It performs the following steps when called:

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](#), throw a **RangeError** exception.
3. Set all of the bytes of *db* to 0.
4. Return *db*.

6.2.8.2 CreateSharedByteDataBlock (*size*)

The abstract operation CreateSharedByteDataBlock takes argument *size* (a non-negative [integer](#)). It performs the following steps when called:

1. **Assert**: *size* ≥ 0.
2. Let *db* be a new [Shared Data Block](#) value consisting of *size* bytes. If it is impossible to create such a [Shared Data Block](#), throw a **RangeError** exception.
3. Let *execution* be the [\[\[CandidateExecution\]\]](#) field of the [surrounding agent's Agent Record](#).
4. Let *eventList* be the [\[\[EventList\]\]](#) field of the element in *execution*.[\[\[EventsRecords\]\]](#) whose [\[\[AgentSignifier\]\]](#) is [AgentSignifier\(\)](#).
5. Let *zero* be « 0 ».
6. For each index *i* of *db*, do
 - a. Append [WriteSharedMemory](#) { [\[\[Order\]\]](#): [Init](#), [\[\[NoTear\]\]](#): **true**, [\[\[Block\]\]](#): *db*, [\[\[ByteIndex\]\]](#): *i*, [\[\[ElementSize\]\]](#): 1, [\[\[Payload\]\]](#): *zero* } to *eventList*.
7. Return *db*.

6.2.8.3 CopyDataBlockBytes (*toBlock*, *toIndex*, *fromBlock*, *fromIndex*, *count*)

The abstract operation CopyDataBlockBytes takes arguments *toBlock*, *toIndex* (a non-negative [integer](#)), *fromBlock*,

fromIndex (a non-negative [integer](#)), and *count* (a non-negative [integer](#)). It performs the following steps when called:

1. **Assert:** *fromBlock* and *toBlock* are distinct [Data Block](#) or [Shared Data Block](#) values.
2. Let *fromSize* be the number of bytes in *fromBlock*.
3. **Assert:** $fromIndex + count \leq fromSize$.
4. Let *toSize* be the number of bytes in *toBlock*.
5. **Assert:** $toIndex + count \leq toSize$.
6. Repeat, while $count > 0$,
 - a. If *fromBlock* is a [Shared Data Block](#), then
 - i. Let *execution* be the `[[CandidateExecution]]` field of the [surrounding agent's Agent Record](#).
 - ii. Let *eventList* be the `[[EventList]]` field of the element in *execution*.`[[EventsRecords]]` whose `[[AgentSignifier]]` is [AgentSignifier\(\)](#).
 - iii. Let *bytes* be a [List](#) whose sole element is a nondeterministically chosen [byte value](#).
 - iv. NOTE: In implementations, *bytes* is the result of a non-atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the [memory model](#) to describe observable behaviour of hardware with weak consistency.
 - v. Let *readEvent* be [ReadSharedMemory](#) { `[[Order]]`: [Unordered](#), `[[NoTear]]`: **true**, `[[Block]]`: *fromBlock*, `[[ByteIndex]]`: *fromIndex*, `[[ElementSize]]`: 1 }.
 - vi. Append *readEvent* to *eventList*.
 - vii. Append [Chosen Value Record](#) { `[[Event]]`: *readEvent*, `[[ChosenValue]]`: *bytes* } to *execution*.`[[ChosenValues]]`.
 - viii. If *toBlock* is a [Shared Data Block](#), then
 1. Append [WriteSharedMemory](#) { `[[Order]]`: [Unordered](#), `[[NoTear]]`: **true**, `[[Block]]`: *toBlock*, `[[ByteIndex]]`: *toIndex*, `[[ElementSize]]`: 1, `[[Payload]]`: *bytes* } to *eventList*.
 - ix. Else,
 1. Set *toBlock*[*toIndex*] to *bytes*[0].
 - b. Else,
 - i. **Assert:** *toBlock* is not a [Shared Data Block](#).
 - ii. Set *toBlock*[*toIndex*] to *fromBlock*[*fromIndex*].
 - c. Set *toIndex* to *toIndex* + 1.
 - d. Set *fromIndex* to *fromIndex* + 1.
 - e. Set *count* to *count* - 1.
7. Return [NormalCompletion](#)(empty).

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

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](#). But no other specification types are used

with these operations.

The BigInt type has no implicit conversions in the ECMAScript language; programmers must call BigInt explicitly to convert values from other types.

7.1.1 ToPrimitive (*input* [, *preferredType*])

The abstract operation ToPrimitive takes argument *input* and optional argument *preferredType*. It 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. It performs the following steps when called:

1. **Assert:** *input* is an ECMAScript language value.
2. If **Type**(*input*) is Object, then
 - a. Let *exoticToPrim* be ? **GetMethod**(*input*, @@toPrimitive).
 - b. If *exoticToPrim* is not **undefined**, then
 - i. If *preferredType* is not present, let *hint* be "default".
 - ii. Else if *preferredType* is **string**, let *hint* be "string".
 - iii. Else,
 1. **Assert:** *preferredType* is number.
 2. Let *hint* be "number".
 - iv. Let *result* be ? **Call**(*exoticToPrim*, *input*, « *hint* »).
 - v. If **Type**(*result*) is not Object, return *result*.
 - vi. Throw a **TypeError** exception.
 - c. If *preferredType* is not present, let *preferredType* be number.
 - d. Return ? **OrdinaryToPrimitive**(*input*, *preferredType*).
3. Return *input*.

NOTE

When ToPrimitive is called with no hint, then it generally behaves as if the hint were number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 21.4.4.45) and Symbol objects (see 20.4.3.5) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were string.

7.1.1.1 OrdinaryToPrimitive (*O*, *hint*)

The abstract operation OrdinaryToPrimitive takes arguments *O* and *hint*. It performs the following steps when called:

1. **Assert:** **Type**(*O*) is Object.
2. **Assert:** *hint* is either **string** or number.
3. If *hint* is **string**, then
 - a. Let *methodNames* be « "toString", "valueOf" ».
4. Else,
 - a. Let *methodNames* be « "valueOf", "toString" ».
5. For each element *name* of *methodNames*, do
 - a. Let *method* be ? **Get**(*O*, *name*).
 - b. If **IsCallable**(*method*) is **true**, then
 - i. Let *result* be ? **Call**(*method*, *O*).
 - ii. If **Type**(*result*) is not Object, return *result*.

6. Throw a **TypeError** exception.

7.1.2 ToBoolean (*argument*)

The abstract operation ToBoolean takes argument *argument*. It converts *argument* to a value of type Boolean according to Table 11:

Table 11: ToBoolean Conversions

Argument Type	Result
Undefined	Return false .
Null	Return false .
Boolean	Return <i>argument</i> .
Number	If <i>argument</i> is $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, or NaN, return false ; otherwise return true .
String	If <i>argument</i> is the empty String (its length is 0), return false ; otherwise return true .
Symbol	Return true .
BigInt	If <i>argument</i> is $0_{\mathbb{Z}}$, return false ; otherwise return true .
Object	Return true .

NOTE An alternate algorithm related to the `[[IsHTMLDDA]]` internal slot is mandated in section B.3.7.1.

7.1.3 ToNumeric (*value*)

The abstract operation ToNumeric takes argument *value*. It returns *value* converted to a Number or a BigInt. It performs the following steps when called:

1. Let *primValue* be ? ToPrimitive(*value*, number).
2. If Type(*primValue*) is BigInt, return *primValue*.
3. Return ? ToNumber(*primValue*).

7.1.4 ToNumber (*argument*)

The abstract operation ToNumber takes argument *argument*. It converts *argument* to a value of type Number according to Table 12:

Table 12: `ToNumber` Conversions

Argument Type	Result
Undefined	Return <code>NaN</code> .
Null	Return <code>+0_F</code> .
Boolean	If <i>argument</i> is true , return <code>1_F</code> . If <i>argument</i> is false , return <code>+0_F</code> .
Number	Return <i>argument</i> (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a TypeError exception.
BigInt	Throw a TypeError exception.
Object	Apply the following steps: <ol style="list-style-type: none"> 1. Let <i>primValue</i> be ? <code>ToPrimitive(argument, number)</code>. 2. Return ? <code>ToNumber(primValue)</code>.

7.1.4.1 `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 (6.1.4). If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of `ToNumber` is `NaN`.

NOTE 1 The terminal symbols of this grammar are all composed of characters in the Unicode Basic Multilingual Plane (BMP). Therefore, the result of `ToNumber` will be `NaN` if the string contains any **leading surrogate** or **trailing surrogate** code units, whether paired or unpaired.

Syntax

```
StringNumericLiteral :::
  StrWhiteSpaceopt
  StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt
```

```
StrWhiteSpace :::
  StrWhiteSpaceChar StrWhiteSpaceopt
```

```
StrWhiteSpaceChar :::
  WhiteSpace
  LineTerminator
```

```
StrNumericLiteral :::
  StrDecimalLiteral
  NonDecimalIntegerLiteral [~Sep]
```

StrDecimalLiteral :::

StrUnsignedDecimalLiteral

+ *StrUnsignedDecimalLiteral*

- *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral :::

Infinity

*DecimalDigits*_[~Sep] . *DecimalDigits*_[~Sep] *opt* *ExponentPart*_[~Sep] *opt*

. *DecimalDigits*_[~Sep] *ExponentPart*_[~Sep] *opt*

*DecimalDigits*_[~Sep] *ExponentPart*_[~Sep] *opt*

All grammar symbols not explicitly defined above have the definitions used in the Lexical Grammar for numeric literals (12.8.3)

NOTE 2

Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral*:

- 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_F**.
- **Infinity** and **-Infinity** are recognized as a *StringNumericLiteral* but not as a *NumericLiteral*.
- A *StringNumericLiteral* cannot include a *BigIntLiteralSuffix*.

7.1.4.1.1 Runtime Semantics: MV

The conversion of a String to a **Number value** is similar overall to the determination of the **Number value** for a numeric literal (see 12.8.3), 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. 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 on any grammar symbol, not provided below, is the MV for that symbol defined in 12.8.3.1.

- The MV of *StringNumericLiteral* ::: [empty] is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace*_{opt} *StrNumericLiteral* *StrWhiteSpace*_{opt} is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *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_F** or **-0_F** as appropriate.)
- The MV of *StrUnsignedDecimalLiteral* ::: **Infinity** is 10^{10000} (a value so large that it will round to **+∞_F**).
- 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 . ExponentPart` is the MV of `DecimalDigits` times 10^e , where e is the MV of `ExponentPart`.
- The MV of `StrUnsignedDecimalLiteral` `::: DecimalDigits . DecimalDigits ExponentPart` is (the MV of the first `DecimalDigits` plus (the MV of the second `DecimalDigits` times 10^{-n})) times 10^e , where n is the number of 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^{e-n} , where n is the number of code points in `DecimalDigits` and e is the MV of `ExponentPart`.
- The MV of `StrUnsignedDecimalLiteral` `::: DecimalDigits ExponentPart` is the MV of `DecimalDigits` times 10^e , where e is the MV of `ExponentPart`.

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 $+0_{\mathbb{F}}$ unless the first non white space code point in the String numeric literal is `-`, in which case the rounded value is $-0_{\mathbb{F}}$. Otherwise, the rounded value must be the **Number value** for the MV (in the sense defined in 6.1.6.1), 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 non-zero digit to its left and there is a non-zero digit, not in the `ExponentPart`, to its right.

7.1.5 ToIntegerOrInfinity (*argument*)

The abstract operation `ToIntegerOrInfinity` takes argument *argument*. It converts *argument* to an **integer**, $+\infty$, or $-\infty$. It performs the following steps when called:

1. Let *number* be ? `ToNumber(argument)`.
2. If *number* is `NaN`, $+0_{\mathbb{F}}$ or $-0_{\mathbb{F}}$, return 0.
3. If *number* is $+\infty_{\mathbb{F}}$, return $+\infty$.
4. If *number* is $-\infty_{\mathbb{F}}$, return $-\infty$.
5. Let *integer* be `floor(abs($\mathbb{R}(number)$))`.
6. If *number* $< +0_{\mathbb{F}}$, set *integer* to $-integer$.
7. Return *integer*.

7.1.6 ToInt32 (*argument*)

The abstract operation `ToInt32` takes argument *argument*. It converts *argument* to one of 2^{32} **integral Number** values in the range $\mathbb{F}(-2^{31})$ through $\mathbb{F}(2^{31} - 1)$, inclusive. It performs the following steps when called:

1. Let *number* be ? `ToNumber(argument)`.
2. If *number* is `NaN`, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
3. Let *int* be the **mathematical value** that is the same sign as *number* and whose magnitude is `floor(abs($\mathbb{R}(number)$))`.
4. Let *int32bit* be *int* modulo 2^{32} .

5. If $int32bit \geq 2^{31}$, return $\mathbb{F}(int32bit - 2^{32})$; otherwise return $\mathbb{F}(int32bit)$.

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 the same value as $ToInt32(x)$ for all values of x . (It is to preserve this latter property that $+\infty_{\mathbb{F}}$ and $-\infty_{\mathbb{F}}$ are mapped to $+0_{\mathbb{F}}$.)
- ToInt32 maps $-0_{\mathbb{F}}$ to $+0_{\mathbb{F}}$.

7.1.7 ToUint32 (*argument*)

The abstract operation ToUint32 takes argument *argument*. It converts *argument* to one of 2^{32} integral Number values in the range $+0_{\mathbb{F}}$ through $\mathbb{F}(2^{32} - 1)$, inclusive. It performs the following steps when called:

1. Let *number* be ? $ToNumber(argument)$.
2. If *number* is NaN, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$ return $+0_{\mathbb{F}}$.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is $\text{floor}(\text{abs}(\mathbb{R}(number)))$.
4. Let *int32bit* be *int* modulo 2^{32} .
5. Return $\mathbb{F}(int32bit)$.

NOTE

Given the above definition of ToUint32:

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

7.1.8 ToInt16 (*argument*)

The abstract operation ToInt16 takes argument *argument*. It converts *argument* to one of 2^{16} integral Number values in the range $\mathbb{F}(-2^{15})$ through $\mathbb{F}(2^{15} - 1)$, inclusive. It performs the following steps when called:

1. Let *number* be ? $ToNumber(argument)$.
2. If *number* is NaN, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$ return $+0_{\mathbb{F}}$.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is $\text{floor}(\text{abs}(\mathbb{R}(number)))$.
4. Let *int16bit* be *int* modulo 2^{16} .
5. If $int16bit \geq 2^{15}$, return $\mathbb{F}(int16bit - 2^{16})$; otherwise return $\mathbb{F}(int16bit)$.

7.1.9 ToUint16 (*argument*)

The abstract operation ToUint16 takes argument *argument*. It converts *argument* to one of 2^{16} integral Number values in the range $+0_{\mathbb{F}}$ through $\mathbb{F}(2^{16} - 1)$, inclusive. It performs the following steps when called:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is NaN, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is $\text{floor}(\text{abs}(\mathbb{R}(\text{number})))$.
4. Let *int16bit* be *int* modulo 2^{16} .
5. Return $\mathbb{F}(\text{int16bit})$.

NOTE

Given the above definition of ToUint16:

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

7.1.10 ToInt8 (*argument*)

The abstract operation ToInt8 takes argument *argument*. It converts *argument* to one of 2^8 integral Number values in the range $-128_{\mathbb{F}}$ through $127_{\mathbb{F}}$, inclusive. It performs the following steps when called:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is NaN, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is $\text{floor}(\text{abs}(\mathbb{R}(\text{number})))$.
4. Let *int8bit* be *int* modulo 2^8 .
5. If *int8bit* $\geq 2^7$, return $\mathbb{F}(\text{int8bit} - 2^8)$; otherwise return $\mathbb{F}(\text{int8bit})$.

7.1.11 ToUint8 (*argument*)

The abstract operation ToUint8 takes argument *argument*. It converts *argument* to one of 2^8 integral Number values in the range $+0_{\mathbb{F}}$ through $255_{\mathbb{F}}$, inclusive. It performs the following steps when called:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is NaN, $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$ or $-\infty_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
3. Let *int* be the mathematical value that is the same sign as *number* and whose magnitude is $\text{floor}(\text{abs}(\mathbb{R}(\text{number})))$.
4. Let *int8bit* be *int* modulo 2^8 .
5. Return $\mathbb{F}(\text{int8bit})$.

7.1.12 ToUint8Clamp (*argument*)

The abstract operation ToUint8Clamp takes argument *argument*. It converts *argument* to one of 2^8 integral Number

values in the range $+0_{\mathbb{F}}$ through $255_{\mathbb{F}}$, inclusive. It performs the following steps when called:

1. Let *number* be ? `ToNumber(argument)`.
2. If *number* is **NaN**, return $+0_{\mathbb{F}}$.
3. If $\mathbb{R}(number) \leq 0$, return $+0_{\mathbb{F}}$.
4. If $\mathbb{R}(number) \geq 255$, return $255_{\mathbb{F}}$.
5. Let *f* be `floor`($\mathbb{R}(number)$).
6. If $f + 0.5 < \mathbb{R}(number)$, return $\mathbb{F}(f + 1)$.
7. If $\mathbb{R}(number) < f + 0.5$, return $\mathbb{F}(f)$.
8. If *f* is odd, return $\mathbb{F}(f + 1)$.
9. Return $\mathbb{F}(f)$.

NOTE

Unlike the other ECMAScript **integer** conversion abstract operation, `ToUint8Clamp` rounds rather than truncates non-integral values and does not convert $+\infty_{\mathbb{F}}$ to $+0_{\mathbb{F}}$. `ToUint8Clamp` does “round half to even” tie-breaking. This differs from **Math.round** which does “round half up” tie-breaking.

7.1.13 ToBigInt (*argument*)

The abstract operation `ToBigInt` takes argument *argument*. It converts *argument* to a `BigInt` value, or throws if an implicit conversion from `Number` would be required. It performs the following steps when called:

1. Let *prim* be ? `ToPrimitive(argument, number)`.
2. Return the value that *prim* corresponds to in [Table 13](#).

Table 13: `BigInt` Conversions

Argument Type	Result
Undefined	Throw a TypeError exception.
Null	Throw a TypeError exception.
Boolean	Return 1n if <i>prim</i> is true and 0n if <i>prim</i> is false .
BigInt	Return <i>prim</i> .
Number	Throw a TypeError exception.
String	<ol style="list-style-type: none"> 1. Let <i>n</i> be ! <code>StringToBigInt(prim)</code>. 2. If <i>n</i> is NaN, throw a SyntaxError exception. 3. Return <i>n</i>.
Symbol	Throw a TypeError exception.

7.1.14 StringToBigInt (*argument*)

Apply the algorithm in [7.1.4.1](#) with the following changes:

- Replace the *StrUnsignedDecimalLiteral* production with *DecimalDigits* to not allow **Infinity**, decimal points, or exponents.
- If the MV is NaN, return NaN, otherwise return the BigInt which exactly corresponds to the MV, rather than rounding to a Number.

7.1.15 ToBigInt64 (*argument*)

The abstract operation ToBigInt64 takes argument *argument*. It converts *argument* to one of 2^{64} BigInt values in the range $\mathbb{Z}(-2^{63})$ through $\mathbb{Z}(2^{63}-1)$, inclusive. It performs the following steps when called:

1. Let n be ? ToBigInt(*argument*).
2. Let *int64bit* be $\mathbb{R}(n)$ modulo 2^{64} .
3. If $\text{int64bit} \geq 2^{63}$, return $\mathbb{Z}(\text{int64bit} - 2^{64})$; otherwise return $\mathbb{Z}(\text{int64bit})$.

7.1.16 ToBigUint64 (*argument*)

The abstract operation ToBigUint64 takes argument *argument*. It converts *argument* to one of 2^{64} BigInt values in the range $0_{\mathbb{Z}}$ through the BigInt value for $\mathbb{Z}(2^{64}-1)$, inclusive. It performs the following steps when called:

1. Let n be ? ToBigInt(*argument*).
2. Let *int64bit* be $\mathbb{R}(n)$ modulo 2^{64} .
3. Return $\mathbb{Z}(\text{int64bit})$.

7.1.17 ToString (*argument*)

The abstract operation ToString takes argument *argument*. It converts *argument* to a value of type String according to [Table 14](#):

Table 14: **ToString** Conversions

Argument Type	Result
Undefined	Return "undefined" .
Null	Return "null" .
Boolean	If <i>argument</i> is true , return "true" . If <i>argument</i> is false , return "false" .
Number	Return ! Number::toString (<i>argument</i>).
String	Return <i>argument</i> .
Symbol	Throw a TypeError exception.
BigInt	Return ! BigInt::toString (<i>argument</i>).
Object	Apply the following steps: <ol style="list-style-type: none"> 1. Let <i>primValue</i> be ? ToPrimitive(<i>argument</i>, string). 2. Return ? ToString(<i>primValue</i>).

7.1.18 ToObject (*argument*)

The abstract operation ToObject takes argument *argument*. It converts *argument* to a value of type Object according to Table 15:

Table 15: **ToObject** Conversions

Argument Type	Result
Undefined	Throw a TypeError exception.
Null	Throw a TypeError exception.
Boolean	Return a new Boolean object whose <code>[[BooleanData]]</code> internal slot is set to <i>argument</i> . See 20.3 for a description of Boolean objects.
Number	Return a new Number object whose <code>[[NumberData]]</code> internal slot is set to <i>argument</i> . See 21.1 for a description of Number objects.
String	Return a new String object whose <code>[[StringData]]</code> internal slot is set to <i>argument</i> . See 22.1 for a description of String objects.
Symbol	Return a new Symbol object whose <code>[[SymbolData]]</code> internal slot is set to <i>argument</i> . See 20.4 for a description of Symbol objects.
BigInt	Return a new BigInt object whose <code>[[BigIntData]]</code> internal slot is set to <i>argument</i> . See 21.2 for a description of BigInt objects.
Object	Return <i>argument</i> .

7.1.19 ToPropertyKey (*argument*)

The abstract operation ToPropertyKey takes argument *argument*. It converts *argument* to a value that can be used as a property key. It performs the following steps when called:

1. Let *key* be ? ToPrimitive(*argument*, string).
2. If Type(*key*) is Symbol, then
 - a. Return *key*.
3. Return ! ToString(*key*).

7.1.20 ToLength (*argument*)

The abstract operation ToLength takes argument *argument*. It converts *argument* to an integral Number suitable for use as the length of an array-like object. It performs the following steps when called:

1. Let *len* be ? ToIntegerOrInfinity(*argument*).
2. If *len* ≤ 0, return +0_F.
3. Return F(min(*len*, 2⁵³ - 1)).

7.1.21 CanonicalNumericIndexString (*argument*)

The abstract operation CanonicalNumericIndexString takes argument *argument*. It returns *argument* converted to a Number value if it is a String representation of a Number that would be produced by ToString, or the string "-0". Otherwise, it returns **undefined**. It performs the following steps when called:

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

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

7.1.22 ToIndex (*value*)

The abstract operation `ToIndex` takes argument *value*. It returns *value* argument converted to a non-negative `integer` if it is a valid `integer index` value. It performs the following steps when called:

1. If *value* is `undefined`, then
 - a. Return `0`.
2. Else,
 - a. Let `integerIndex` be `F(? ToIntegerOrInfinity(value))`.
 - b. If `integerIndex < +0F`, throw a `RangeError` exception.
 - c. Let `index` be `! ToLength(integerIndex)`.
 - d. If `! SameValue(integerIndex, index)` is `false`, throw a `RangeError` exception.
 - e. Return `R(index)`.

7.2 Testing and Comparison Operations

7.2.1 RequireObjectCoercible (*argument*)

The abstract operation `RequireObjectCoercible` takes argument *argument*. It throws an error if *argument* is a value that cannot be converted to an `Object` using `ToObject`. It is defined by [Table 16](#):

Table 16: `RequireObjectCoercible` Results

Argument Type	Result
Undefined	Throw a <code>TypeError</code> exception.
Null	Throw a <code>TypeError</code> exception.
Boolean	Return <i>argument</i> .
Number	Return <i>argument</i> .
String	Return <i>argument</i> .
Symbol	Return <i>argument</i> .
BigInt	Return <i>argument</i> .
Object	Return <i>argument</i> .

7.2.2 IsArray (*argument*)

The abstract operation IsArray takes argument *argument*. It performs the following steps when called:

1. If `Type(argument)` is not Object, return **false**.
2. If *argument* is an [Array exotic object](#), return **true**.
3. If *argument* is a [Proxy exotic object](#), then
 - a. If `argument.[[ProxyHandler]]` is **null**, throw a **TypeError** exception.
 - b. Let *target* be `argument.[[ProxyTarget]]`.
 - c. Return ? `IsArray(target)`.
4. Return **false**.

7.2.3 IsCallable (*argument*)

The abstract operation IsCallable takes argument *argument* (an [ECMAScript language value](#)). It determines if *argument* is a callable function with a `[[Call]]` internal method. It performs the following steps when called:

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

7.2.4 IsConstructor (*argument*)

The abstract operation IsConstructor takes argument *argument* (an [ECMAScript language value](#)). It determines if *argument* is a [function object](#) with a `[[Construct]]` internal method. It performs the following steps when called:

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

7.2.5 IsExtensible (*O*)

The abstract operation IsExtensible takes argument *O* (an Object) and returns a completion record which, if its `[[Type]]` is normal, has a `[[Value]]` which is a Boolean. It is used to determine whether additional properties can be added to *O*. It performs the following steps when called:

1. **Assert:** `Type(O)` is Object.
2. Return ? `O.[[IsExtensible]]()`.

7.2.6 IsIntegralNumber (*argument*)

The abstract operation IsIntegralNumber takes argument *argument*. It determines if *argument* is a finite [integral Number](#) value. It performs the following steps when called:

1. If `Type(argument)` is not Number, return **false**.
2. If *argument* is NaN, $+\infty_{\mathbb{F}}$, or $-\infty_{\mathbb{F}}$, return **false**.
3. If `floor(abs($\mathbb{R}(argument)$))` \neq `abs($\mathbb{R}(argument)$)`, return **false**.
4. Return **true**.

7.2.7 IsPropertyKey (*argument*)

The abstract operation IsPropertyKey takes argument *argument* (an [ECMAScript language value](#)). It determines if *argument* is a value that may be used as a property key. It performs the following steps when called:

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

7.2.8 IsRegExp (*argument*)

The abstract operation IsRegExp takes argument *argument*. It performs the following steps when called:

1. If `Type(argument)` is not Object, return **false**.
2. Let *matcher* be `? Get(argument, @@match)`.
3. If *matcher* is not **undefined**, return `! ToBoolean(matcher)`.
4. If *argument* has a `[[RegExpMatcher]]` internal slot, return **true**.
5. Return **false**.

7.2.9 IsStringPrefix (*p*, *q*)

The abstract operation IsStringPrefix takes arguments *p* (a String) and *q* (a String). It determines if *p* is a prefix of *q*. It performs the following steps when called:

1. **Assert:** `Type(p)` is String.
2. **Assert:** `Type(q)` is String.
3. If *q* can be the [string-concatenation](#) of *p* and some other String *r*, return **true**. Otherwise, return **false**.

NOTE Any String is a prefix of itself, because *r* may be the empty String.

7.2.10 SameValue (*x*, *y*)

The abstract operation SameValue takes arguments *x* (an [ECMAScript language value](#)) and *y* (an [ECMAScript language value](#)) and returns a completion record whose `[[Type]]` is **normal** and whose `[[Value]]` is a Boolean. It performs the following steps when called:

1. If `Type(x)` is different from `Type(y)`, return **false**.
2. If `Type(x)` is Number or BigInt, then
 - a. Return `! Type(x)::sameValue(x, y)`.
3. Return `! SameValueNonNumeric(x, y)`.

NOTE This algorithm differs from the [Strict Equality Comparison](#) Algorithm in its treatment of signed zeroes and NaNs.

7.2.11 SameValueZero (*x*, *y*)

The abstract operation SameValueZero takes arguments *x* (an [ECMAScript language value](#)) and *y* (an [ECMAScript](#)

language value) and returns a completion record whose [[Type]] is normal and whose [[Value]] is a Boolean. It performs the following steps when called:

1. If `Type(x)` is different from `Type(y)`, return **false**.
2. If `Type(x)` is Number or BigInt, then
 - a. Return ! `Type(x)::sameValueZero(x, y)`.
3. Return ! `SameValueNonNumeric(x, y)`.

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

7.2.12 SameValueNonNumeric (*x*, *y*)

The abstract operation `SameValueNonNumeric` takes arguments *x* (an ECMAScript language value) and *y* (an ECMAScript language value) and returns a completion record whose [[Type]] is normal and whose [[Value]] is a Boolean. It performs the following steps when called:

1. Assert: `Type(x)` is not Number or BigInt.
2. Assert: `Type(x)` is the same as `Type(y)`.
3. If `Type(x)` is Undefined, return **true**.
4. If `Type(x)` is Null, return **true**.
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 at corresponding indices), return **true**; otherwise, return **false**.
6. If `Type(x)` is Boolean, then
 - a. If *x* and *y* are both **true** or both **false**, return **true**; otherwise, return **false**.
7. If `Type(x)` is Symbol, then
 - a. If *x* and *y* are both the same Symbol value, return **true**; otherwise, return **false**.
8. If *x* and *y* are the same Object value, return **true**. Otherwise, return **false**.

7.2.13 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 *LeftFirst* as a parameter. The flag is used to control the order in which operations with potentially visible side-effects are performed upon *x* and *y*. It is necessary because ECMAScript specifies left to right evaluation of expressions. The default value of *LeftFirst* is **true** and indicates that the *x* parameter corresponds to an expression that occurs to the left of the *y* parameter's corresponding expression. If *LeftFirst* is **false**, the reverse is the case and operations must be performed upon *y* before *x*. Such a comparison is performed as follows:

1. If the *LeftFirst* flag is **true**, then
 - a. Let *px* be ? `ToPrimitive(x, number)`.
 - b. Let *py* be ? `ToPrimitive(y, number)`.
2. Else,
 - a. NOTE: The order of evaluation needs to be reversed to preserve left to right evaluation.
 - b. Let *py* be ? `ToPrimitive(y, number)`.
 - c. Let *px* be ? `ToPrimitive(x, number)`.
3. If `Type(px)` is String and `Type(py)` is String, then

- a. If `IsStringPrefix(py, px)` is **true**, return **false**.
 - b. If `IsStringPrefix(px, py)` is **true**, return **true**.
 - c. Let k be the smallest non-negative integer such that the code unit at index k within px is different from the code unit at index 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 numeric value of the code unit at index k within px .
 - e. Let n be the integer that is the numeric value of the code unit at index k within py .
 - f. If $m < n$, return **true**. Otherwise, return **false**.
4. Else,
- a. If `Type(px)` is `BigInt` and `Type(py)` is `String`, then
 - i. Let ny be `!StringToBigInt(py)`.
 - ii. If ny is `NaN`, return **undefined**.
 - iii. Return `BigInt::lessThan(px, ny)`.
 - b. If `Type(px)` is `String` and `Type(py)` is `BigInt`, then
 - i. Let nx be `!StringToBigInt(px)`.
 - ii. If nx is `NaN`, return **undefined**.
 - iii. Return `BigInt::lessThan(nx, py)`.
 - c. NOTE: Because px and py are primitive values, evaluation order is not important.
 - d. Let nx be `!ToNumeric(px)`.
 - e. Let ny be `!ToNumeric(py)`.
 - f. If `Type(nx)` is the same as `Type(ny)`, return `Type(nx)::lessThan(nx, ny)`.
 - g. **Assert**: `Type(nx)` is `BigInt` and `Type(ny)` is `Number`, or `Type(nx)` is `Number` and `Type(ny)` is `BigInt`.
 - h. If nx or ny is `NaN`, return **undefined**.
 - i. If nx is $-\infty_{\mathbb{F}}$ or ny is $+\infty_{\mathbb{F}}$, return **true**.
 - j. If nx is $+\infty_{\mathbb{F}}$ or ny is $-\infty_{\mathbb{F}}$, return **false**.
 - k. If $\mathbb{R}(nx) < \mathbb{R}(ny)$, return **true**; otherwise return **false**.

NOTE 1 Step 3 differs from step 2.c in the algorithm that handles the addition operator `+` (13.15.3) by using the logical-and operation instead of the logical-or operation.

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.

7.2.14 Abstract Equality Comparison

The comparison `x == y`, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If `Type(x)` is the same as `Type(y)`, then
 - a. Return the result of performing `Strict Equality Comparison x === y`.
2. If x is **null** and y is **undefined**, return **true**.
3. If x is **undefined** and y is **null**, return **true**.
4. NOTE: This step is replaced in section B.3.7.2.

5. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x === ! ToNumber(y)`.
6. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `! ToNumber(x) === y`.
7. If `Type(x)` is `BigInt` and `Type(y)` is `String`, then
 - a. Let `n` be `! StringToBigInt(y)`.
 - b. If `n` is `NaN`, return `false`.
 - c. Return the result of the comparison `x === n`.
8. If `Type(x)` is `String` and `Type(y)` is `BigInt`, return the result of the comparison `y === x`.
9. If `Type(x)` is `Boolean`, return the result of the comparison `! ToNumber(x) === y`.
10. If `Type(y)` is `Boolean`, return the result of the comparison `x === ! ToNumber(y)`.
11. If `Type(x)` is either `String`, `Number`, `BigInt`, or `Symbol` and `Type(y)` is `Object`, return the result of the comparison `x === ? ToPrimitive(y)`.
12. If `Type(x)` is `Object` and `Type(y)` is either `String`, `Number`, `BigInt`, or `Symbol`, return the result of the comparison `? ToPrimitive(x) === y`.
13. If `Type(x)` is `BigInt` and `Type(y)` is `Number`, or if `Type(x)` is `Number` and `Type(y)` is `BigInt`, then
 - a. If `x` or `y` are any of `NaN`, $+\infty_{\mathbb{F}}$, or $-\infty_{\mathbb{F}}$, return `false`.
 - b. If $\mathbb{R}(x) = \mathbb{R}(y)$, return `true`; otherwise return `false`.
14. Return `false`.

7.2.15 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 `Number` or `BigInt`, then
 - a. Return `! Type(x)::equal(x, y)`.
3. Return `! SameValueNonNumeric(x, y)`.

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

7.3 Operations on Objects

7.3.1 MakeBasicObject (*internalSlotsList*)

The abstract operation `MakeBasicObject` takes argument *internalSlotsList*. It is the source of all ECMAScript objects that are created algorithmically, including both ordinary objects and exotic objects. It factors out common steps used in creating all objects, and centralizes object creation. It performs the following steps when called:

1. **Assert:** *internalSlotsList* is a `List` of internal slot names.
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 10.1.
4. **Assert:** If the caller will not be overriding both *obj*'s `[[GetPrototypeOf]]` and `[[SetPrototypeOf]]` essential internal methods, then *internalSlotsList* contains `[[Prototype]]`.
5. **Assert:** If the caller will not be overriding all of *obj*'s `[[SetPrototypeOf]]`, `[[IsExtensible]]`, and `[[PreventExtensions]]` essential internal methods, then *internalSlotsList* contains `[[Extensible]]`.
6. If *internalSlotsList* contains `[[Extensible]]`, set *obj*.`[[Extensible]]` to `true`.
7. Return *obj*.

NOTE

Within this specification, exotic objects are created in [abstract operations](#) such as [ArrayCreate](#) and [BoundFunctionCreate](#) by first calling [MakeBasicObject](#) to obtain a basic, foundational object, and then overriding some or all of that object's internal methods. In order to encapsulate [exotic object](#) creation, the object's essential internal methods are never modified outside those operations.

7.3.2 Get (*O*, *P*)

The abstract operation [Get](#) takes arguments *O* (an Object) and *P* (a property key). It is used to retrieve the value of a specific property of an object. It performs the following steps when called:

1. [Assert](#): [Type](#)(*O*) is Object.
2. [Assert](#): [IsPropertyKey](#)(*P*) is **true**.
3. Return ? *O*.[[Get]](*P*, *O*).

7.3.3 GetV (*V*, *P*)

The abstract operation [GetV](#) takes arguments *V* (an [ECMAScript language value](#)) and *P* (a property key). It is used to retrieve the value of a specific property of an [ECMAScript language value](#). If the value is not an object, the property lookup is performed using a wrapper object appropriate for the type of the value. It performs the following steps when called:

1. [Assert](#): [IsPropertyKey](#)(*P*) is **true**.
2. Let *O* be ? [ToObject](#)(*V*).
3. Return ? *O*.[[Get]](*P*, *V*).

7.3.4 Set (*O*, *P*, *V*, *Throw*)

The abstract operation [Set](#) takes arguments *O* (an Object), *P* (a property key), *V* (an [ECMAScript language value](#)), and *Throw* (a Boolean). It is used to set the value of a specific property of an object. *V* is the new value for the property. It performs the following steps when called:

1. [Assert](#): [Type](#)(*O*) is Object.
2. [Assert](#): [IsPropertyKey](#)(*P*) is **true**.
3. [Assert](#): [Type](#)(*Throw*) is Boolean.
4. Let *success* be ? *O*.[[Set]](*P*, *V*, *O*).
5. If *success* is **false** and *Throw* is **true**, throw a [TypeError](#) exception.
6. Return *success*.

7.3.5 CreateDataProperty (*O*, *P*, *V*)

The abstract operation [CreateDataProperty](#) takes arguments *O* (an Object), *P* (a property key), and *V* (an [ECMAScript language value](#)). It is used to create a new own property of an object. It performs the following steps when called:

1. [Assert](#): [Type](#)(*O*) is Object.
2. [Assert](#): [IsPropertyKey](#)(*P*) is **true**.
3. Let *newDesc* be the [PropertyDescriptor](#) { [[Value]]: *V*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true** }.

4. Return ? $O.[[\text{DefineOwnProperty}]](P, \text{newDesc})$.

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 if O is not extensible, $[[\text{DefineOwnProperty}]]$ will return **false**.

7.3.6 CreateMethodProperty (O, P, V)

The abstract operation CreateMethodProperty takes arguments O (an Object), P (a property key), and V (an ECMAScript language value). It is used to create a new own property of an object. It performs the following steps when called:

1. **Assert:** $\text{Type}(O)$ is Object.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. Let newDesc be the PropertyDescriptor { $[[\text{Value}]]: V$, $[[\text{Writable}]]: \text{true}$, $[[\text{Enumerable}]]: \text{false}$, $[[\text{Configurable}]]: \text{true}$ }.
4. Return ? $O.[[\text{DefineOwnProperty}]](P, \text{newDesc})$.

NOTE This abstract operation creates a property whose attributes are set to the same defaults used for built-in methods and methods defined using class declaration syntax. Normally, the property will not already exist. If it does exist and is not configurable or if O is not extensible, $[[\text{DefineOwnProperty}]]$ will return **false**.

7.3.7 CreateDataPropertyOrThrow (O, P, V)

The abstract operation CreateDataPropertyOrThrow takes arguments O (an Object), P (a property key), and V (an ECMAScript language value). It is used to create a new own property of an object. It throws a **TypeError** exception if the requested property update cannot be performed. It performs the following steps when called:

1. **Assert:** $\text{Type}(O)$ is Object.
2. **Assert:** $\text{IsPropertyKey}(P)$ is **true**.
3. Let success be ? $\text{CreateDataProperty}(O, P, V)$.
4. If success is **false**, throw a **TypeError** exception.
5. Return success .

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 if O is not extensible, $[[\text{DefineOwnProperty}]]$ will return **false** causing this operation to throw a **TypeError** exception.

7.3.8 DefinePropertyOrThrow (O, P, desc)

The abstract operation DefinePropertyOrThrow takes arguments O (an Object), P (a property key), and desc (a Property Descriptor). It is used to call the $[[\text{DefineOwnProperty}]]$ internal method of an object in a manner that will throw a **TypeError** exception if the requested property update cannot be performed. It performs the following steps

when called:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is **true**.
3. Let *success* be ? `O.[[DefineOwnProperty]](P, desc)`.
4. If *success* is **false**, throw a **TypeError** exception.
5. Return *success*.

7.3.9 DeletePropertyOrThrow (*O*, *P*)

The abstract operation DeletePropertyOrThrow takes arguments *O* (an Object) and *P* (a property key). It is used to remove a specific own property of an object. It throws an exception if the property is not configurable. It performs the following steps when called:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is **true**.
3. Let *success* be ? `O.[[Delete]](P)`.
4. If *success* is **false**, throw a **TypeError** exception.
5. Return *success*.

7.3.10 GetMethod (*V*, *P*)

The abstract operation GetMethod takes arguments *V* (an [ECMAScript language value](#)) and *P* (a property key). It is used to get the value of a specific property of an [ECMAScript language value](#) when the value of the property is expected to be a function. It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *func* be ? `GetV(V, P)`.
3. If *func* is either **undefined** or **null**, return **undefined**.
4. If `IsCallable(func)` is **false**, throw a **TypeError** exception.
5. Return *func*.

7.3.11 HasProperty (*O*, *P*)

The abstract operation HasProperty takes arguments *O* (an Object) and *P* (a property key) and returns a completion record which, if its `[[Type]]` is **normal**, has a `[[Value]]` which is a Boolean. It is used to determine whether an object has a property with the specified property key. The property may be either an own or inherited. It performs the following steps when called:

1. **Assert:** `Type(O)` is Object.
2. **Assert:** `IsPropertyKey(P)` is **true**.
3. Return ? `O.[[HasProperty]](P)`.

7.3.12 HasOwnProperty (*O*, *P*)

The abstract operation HasOwnProperty takes arguments *O* (an Object) and *P* (a property key) and returns a completion record which, if its `[[Type]]` is **normal**, has a `[[Value]]` which is a Boolean. It is used to determine whether an object has an own property with the specified property key. It performs the following steps when called:

1. Assert: `Type(O)` is Object.
2. Assert: `IsPropertyKey(P)` is **true**.
3. Let *desc* be ? `O.[[GetOwnProperty]](P)`.
4. If *desc* is **undefined**, return **false**.
5. Return **true**.

7.3.13 Call (*F*, *V* [, *argumentsList*])

The abstract operation Call takes arguments *F* (an ECMAScript language value) and *V* (an ECMAScript language value) and optional argument *argumentsList* (a List of ECMAScript language values). It is used to call the `[[Call]]` internal method of a function object. *F* is the function object, *V* is an ECMAScript language value that is the **this** value of the `[[Call]]`, and *argumentsList* is the value passed to the corresponding argument of the internal method. If *argumentsList* is not present, a new empty List is used as its value. It performs the following steps when called:

1. If *argumentsList* is not present, set *argumentsList* to a new empty List.
2. If `IsCallable(F)` is **false**, throw a **TypeError** exception.
3. Return ? `F.[[Call]](V, argumentsList)`.

7.3.14 Construct (*F* [, *argumentsList* [, *newTarget*]])

The abstract operation Construct takes argument *F* (a function object) and optional arguments *argumentsList* and *newTarget*. It is used to call the `[[Construct]]` internal method of a function object. *argumentsList* and *newTarget* are the values to be passed as the corresponding arguments of the internal method. If *argumentsList* is not present, a new empty List is used as its value. If *newTarget* is not present, *F* is used as its value. It performs the following steps when called:

1. If *newTarget* is not present, set *newTarget* to *F*.
2. If *argumentsList* is not present, set *argumentsList* to a new empty List.
3. Assert: `IsConstructor(F)` is **true**.
4. Assert: `IsConstructor(newTarget)` is **true**.
5. Return ? `F.[[Construct]](argumentsList, newTarget)`.

NOTE If *newTarget* is not present, this operation is equivalent to: **new F(...argumentsList)**

7.3.15 SetIntegrityLevel (*O*, *level*)

The abstract operation SetIntegrityLevel takes arguments *O* and *level*. It is used to fix the set of own properties of an object. It performs the following steps when called:

1. Assert: `Type(O)` is Object.
2. Assert: *level* is either **sealed** or **frozen**.
3. Let *status* be ? `O.[[PreventExtensions]]()`.
4. If *status* is **false**, return **false**.
5. Let *keys* be ? `O.[[OwnPropertyKeys]]()`.
6. If *level* is **sealed**, then
 - a. For each element *k* of *keys*, do
 - i. Perform ? `DefinePropertyOrThrow(O, k, PropertyDescriptor { [[Configurable]]: false })`.

7. Else,
 - a. **Assert:** *level* is frozen.
 - b. For each element *k* of *keys*, do
 - i. Let *currentDesc* be ? *O*.[[GetOwnProperty]](*k*).
 - ii. If *currentDesc* is not **undefined**, then
 1. If **IsAccessorDescriptor**(*currentDesc*) is **true**, then
 - a. Let *desc* be the PropertyDescriptor { [[Configurable]]: **false** }.
 2. Else,
 - a. Let *desc* be the PropertyDescriptor { [[Configurable]]: **false**, [[Writable]]: **false** }.
 3. Perform ? **DefinePropertyOrThrow**(*O*, *k*, *desc*).
8. Return **true**.

7.3.16 TestIntegrityLevel (*O*, *level*)

The abstract operation TestIntegrityLevel takes arguments *O* and *level*. It is used to determine if the set of own properties of an object are fixed. It performs the following steps when called:

1. **Assert:** **Type**(*O*) is Object.
2. **Assert:** *level* is either **sealed** or **frozen**.
3. Let *extensible* be ? **IsExtensible**(*O*).
4. If *extensible* is **true**, return **false**.
5. NOTE: If the object is extensible, none of its properties are examined.
6. Let *keys* be ? *O*.[[OwnPropertyKeys]]().
7. For each element *k* of *keys*, do
 - a. Let *currentDesc* be ? *O*.[[GetOwnProperty]](*k*).
 - b. If *currentDesc* is not **undefined**, then
 - i. If *currentDesc*.[[Configurable]] is **true**, return **false**.
 - ii. If *level* is **frozen** and **IsDataDescriptor**(*currentDesc*) is **true**, then
 1. If *currentDesc*.[[Writable]] is **true**, return **false**.
8. Return **true**.

7.3.17 CreateArrayFromList (*elements*)

The abstract operation CreateArrayFromList takes argument *elements* (a **List**). It is used to create an Array object whose elements are provided by *elements*. It performs the following steps when called:

1. **Assert:** *elements* is a **List** whose elements are all ECMAScript language values.
2. Let *array* be ! **ArrayCreate**(0).
3. Let *n* be 0.
4. For each element *e* of *elements*, do
 - a. Perform ! **CreateDataPropertyOrThrow**(*array*, ! **ToString**(**F**(*n*)), *e*).
 - b. Set *n* to *n* + 1.
5. Return *array*.

7.3.18 LengthOfArrayLike (*obj*)

The abstract operation LengthOfArrayLike takes argument *obj*. It returns the value of the "**length**" property of an array-like object (as a non-negative **integer**). It performs the following steps when called:

1. Assert: `Type(obj)` is Object.
2. Return `ℝ(? ToLength(? Get(obj, "length")))`.

An *array-like object* is any object for which this operation returns an **integer** rather than an **abrupt completion**.

NOTE 1 Typically, an array-like object would also have some properties with **integer index** names. However, that is not a requirement of this definition.

NOTE 2 Array objects and String objects are examples of array-like objects.

7.3.19 CreateListFromArrayLike (*obj* [, *elementTypes*])

The abstract operation `CreateListFromArrayLike` takes argument *obj* and optional argument *elementTypes* (a **List** of names of ECMAScript Language Types). It is used to create a **List** value whose elements are provided by the indexed properties of *obj*. *elementTypes* contains the names of ECMAScript Language Types that are allowed for element values of the **List** that is created. It performs the following steps when called:

1. If *elementTypes* is not present, set *elementTypes* to « Undefined, Null, Boolean, String, Symbol, Number, BigInt, Object ».
2. If `Type(obj)` is not Object, throw a **TypeError** exception.
3. Let *len* be `? LengthOfArrayLike(obj)`.
4. Let *list* be a new empty **List**.
5. Let *index* be 0.
6. Repeat, while *index* < *len*,
 - a. Let *indexName* be `! ToString(F(index))`.
 - b. Let *next* be `? Get(obj, indexName)`.
 - c. If `Type(next)` is not an element of *elementTypes*, throw a **TypeError** exception.
 - d. Append *next* as the last element of *list*.
 - e. Set *index* to *index* + 1.
7. Return *list*.

7.3.20 Invoke (*V*, *P* [, *argumentsList*])

The abstract operation `Invoke` takes arguments *V* (an **ECMAScript language value**) and *P* (a property key) and optional argument *argumentsList* (a **List** of ECMAScript language values). It is used to call a method property of an **ECMAScript language value**. *V* serves as both the lookup point for the property and the **this** value of the call. *argumentsList* is the list of arguments values passed to the method. If *argumentsList* is not present, a new empty **List** is used as its value. It performs the following steps when called:

1. Assert: `IsPropertyKey(P)` is **true**.
2. If *argumentsList* is not present, set *argumentsList* to a new empty **List**.
3. Let *func* be `? GetV(V, P)`.
4. Return `? Call(func, V, argumentsList)`.

7.3.21 OrdinaryHasInstance (*C*, *O*)

The abstract operation `OrdinaryHasInstance` takes arguments *C* (an **ECMAScript language value**) and *O*. It

implements the default algorithm for determining if *O* inherits from the instance object inheritance path provided by *C*. It performs the following steps when called:

1. If `IsCallable(C)` is **false**, return **false**.
2. If *C* has a `[[BoundTargetFunction]]` internal slot, then
 - a. Let *BC* be `C.[[BoundTargetFunction]]`.
 - b. Return `? InstanceofOperator(O, BC)`.
3. If `Type(O)` is not `Object`, return **false**.
4. Let *P* be `? Get(C, "prototype")`.
5. If `Type(P)` is not `Object`, throw a **TypeError** exception.
6. Repeat,
 - a. Set *O* to `? O.[[GetPrototypeOf]]()`.
 - b. If *O* is **null**, return **false**.
 - c. If `SameValue(P, O)` is **true**, return **true**.

7.3.22 SpeciesConstructor (*O*, *defaultConstructor*)

The abstract operation `SpeciesConstructor` takes arguments *O* (an `Object`) and *defaultConstructor* (a `constructor`). It is used to retrieve the `constructor` that should be used to create new objects that are derived from *O*. *defaultConstructor* is the `constructor` to use if a `constructor @@species` property cannot be found starting from *O*. It performs the following steps when called:

1. **Assert**: `Type(O)` is `Object`.
2. Let *C* be `? Get(O, "constructor")`.
3. If *C* is **undefined**, return *defaultConstructor*.
4. If `Type(C)` is not `Object`, throw a **TypeError** exception.
5. Let *S* be `? Get(C, @@species)`.
6. If *S* is either **undefined** or **null**, return *defaultConstructor*.
7. If `IsConstructor(S)` is **true**, return *S*.
8. Throw a **TypeError** exception.

7.3.23 EnumerableOwnPropertyNames (*O*, *kind*)

The abstract operation `EnumerableOwnPropertyNames` takes arguments *O* (an `Object`) and *kind* (one of `key`, `value`, or `key+value`). It performs the following steps when called:

1. **Assert**: `Type(O)` is `Object`.
2. Let *ownKeys* be `? O.[[OwnPropertyKeys]]()`.
3. Let *properties* be a new empty `List`.
4. For each element *key* of *ownKeys*, do
 - a. If `Type(key)` is `String`, then
 - i. Let *desc* be `? O.[[GetOwnProperty]](key)`.
 - ii. If *desc* is not **undefined** and *desc*.`[[Enumerable]]` is **true**, then
 1. If *kind* is `key`, append *key* to *properties*.
 2. Else,
 - a. Let *value* be `? Get(O, key)`.
 - b. If *kind* is `value`, append *value* to *properties*.
 - c. Else,

- i. **Assert:** *kind* is key+value.
- ii. Let *entry* be ! **CreateArrayFromList**(« *key*, *value* »).
- iii. Append *entry* to *properties*.

5. Return *properties*.

7.3.24 GetFunctionRealm (*obj*)

The abstract operation GetFunctionRealm takes argument *obj*. It performs the following steps when called:

1. **Assert:** ! **IsCallable**(*obj*) is **true**.
2. If *obj* has a **[[Realm]]** internal slot, then
 - a. Return *obj*.**[[Realm]]**.
3. If *obj* is a **bound function exotic object**, then
 - a. Let *target* be *obj*.**[[BoundTargetFunction]]**.
 - b. Return ? **GetFunctionRealm**(*target*).
4. If *obj* is a **Proxy exotic object**, then
 - a. If *obj*.**[[ProxyHandler]]** is **null**, throw a **TypeError** exception.
 - b. Let *proxyTarget* be *obj*.**[[ProxyTarget]]**.
 - c. Return ? **GetFunctionRealm**(*proxyTarget*).
5. Return the **current Realm Record**.

NOTE Step 5 will only be reached if *obj* is a non-standard function **exotic object** that does not have a **[[Realm]]** internal slot.

7.3.25 CopyDataProperties (*target*, *source*, *excludedItems*)

The abstract operation CopyDataProperties takes arguments *target*, *source*, and *excludedItems*. It performs the following steps when called:

1. **Assert:** **Type**(*target*) is Object.
2. **Assert:** *excludedItems* is a **List** of property keys.
3. If *source* is **undefined** or **null**, return *target*.
4. Let *from* be ! **ToObject**(*source*).
5. Let *keys* be ? *from*.**[[OwnPropertyKeys]]**().
6. For each element *nextKey* of *keys*, do
 - a. Let *excluded* be **false**.
 - b. For each element *e* of *excludedItems*, do
 - i. If **SameValue**(*e*, *nextKey*) is **true**, then
 1. Set *excluded* to **true**.
 - c. If *excluded* is **false**, then
 - i. Let *desc* be ? *from*.**[[GetOwnProperty]]**(*nextKey*).
 - ii. If *desc* is not **undefined** and *desc*.**[[Enumerable]]** is **true**, then
 1. Let *propValue* be ? **Get**(*from*, *nextKey*).
 2. Perform ! **CreateDataPropertyOrThrow**(*target*, *nextKey*, *propValue*).
7. Return *target*.

The target passed in here is always a newly created object which is not directly accessible in case of an error being thrown.

7.4 Operations on Iterator Objects

See Common Iteration Interfaces (27.1).

7.4.1 GetIterator (*obj* [, *hint* [, *method*]])

The abstract operation GetIterator takes argument *obj* and optional arguments *hint* and *method*. It performs the following steps when called:

1. If *hint* is not present, set *hint* to *sync*.
2. **Assert:** *hint* is either *sync* or *async*.
3. If *method* is not present, then
 - a. If *hint* is *async*, then
 - i. Set *method* to ? [GetMethod](#)(*obj*, @@*asyncIterator*).
 - ii. If *method* is **undefined**, then
 1. Let *syncMethod* be ? [GetMethod](#)(*obj*, @@*iterator*).
 2. Let *syncIteratorRecord* be ? [GetIterator](#)(*obj*, *sync*, *syncMethod*).
 3. Return ! [CreateAsyncFromSyncIterator](#)(*syncIteratorRecord*).
 - b. Otherwise, set *method* to ? [GetMethod](#)(*obj*, @@*iterator*).
4. Let *iterator* be ? [Call](#)(*method*, *obj*).
5. If [Type](#)(*iterator*) is not **Object**, throw a **TypeError** exception.
6. Let *nextMethod* be ? [GetV](#)(*iterator*, "next").
7. Let *iteratorRecord* be the [Record](#) { [[*Iterator*]]: *iterator*, [[*NextMethod*]]: *nextMethod*, [[*Done*]]: **false** }.
8. Return *iteratorRecord*.

7.4.2 IteratorNext (*iteratorRecord* [, *value*])

The abstract operation IteratorNext takes argument *iteratorRecord* and optional argument *value*. It performs the following steps when called:

1. If *value* is not present, then
 - a. Let *result* be ? [Call](#)(*iteratorRecord*.[[*NextMethod*]], *iteratorRecord*.[[*Iterator*]]).
2. Else,
 - a. Let *result* be ? [Call](#)(*iteratorRecord*.[[*NextMethod*]], *iteratorRecord*.[[*Iterator*]], « *value* »).
3. If [Type](#)(*result*) is not **Object**, throw a **TypeError** exception.
4. Return *result*.

7.4.3 IteratorComplete (*iterResult*)

The abstract operation IteratorComplete takes argument *iterResult*. It performs the following steps when called:

1. **Assert:** [Type](#)(*iterResult*) is **Object**.
2. Return ! [ToBoolean](#)(? [Get](#)(*iterResult*, "done")).

7.4.4 IteratorValue (*iterResult*)

The abstract operation IteratorValue takes argument *iterResult*. It performs the following steps when called:

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

7.4.5 IteratorStep (*iteratorRecord*)

The abstract operation IteratorStep takes argument *iteratorRecord*. It requests the next value from *iteratorRecord*.
[[Iterator]] by calling *iteratorRecord*.[[NextMethod]] and returns either **false** indicating that the iterator has reached its end or the IteratorResult object if a next value is available. It performs the following steps when called:

1. Let *result* be ? `IteratorNext(iteratorRecord)`.
2. Let *done* be ? `IteratorComplete(result)`.
3. If *done* is **true**, return **false**.
4. Return *result*.

7.4.6 IteratorClose (*iteratorRecord, completion*)

The abstract operation IteratorClose takes arguments *iteratorRecord* and *completion*. It is used to notify an iterator that it should perform any actions it would normally perform when it has reached its completed state. It performs the following steps when called:

1. Assert: `Type(iteratorRecord. [[Iterator]])` is Object.
2. Assert: *completion* is a Completion Record.
3. Let *iterator* be *iteratorRecord*.[[Iterator]].
4. Let *innerResult* be `GetMethod(iterator, "return")`.
5. If *innerResult*.[[Type]] is normal, then
 - a. Let *return* be *innerResult*.[[Value]].
 - b. If *return* is **undefined**, return `Completion(completion)`.
 - c. Set *innerResult* to `Call(return, iterator)`.
6. If *completion*.[[Type]] is throw, return `Completion(completion)`.
7. If *innerResult*.[[Type]] is throw, return `Completion(innerResult)`.
8. If `Type(innerResult. [[Value]])` is not Object, throw a **TypeError** exception.
9. Return `Completion(completion)`.

7.4.7 AsyncIteratorClose (*iteratorRecord, completion*)

The abstract operation AsyncIteratorClose takes arguments *iteratorRecord* and *completion*. It is used to notify an async iterator that it should perform any actions it would normally perform when it has reached its completed state. It performs the following steps when called:

1. Assert: `Type(iteratorRecord. [[Iterator]])` is Object.
2. Assert: *completion* is a Completion Record.
3. Let *iterator* be *iteratorRecord*.[[Iterator]].
4. Let *innerResult* be `GetMethod(iterator, "return")`.
5. If *innerResult*.[[Type]] is normal, then
 - a. Let *return* be *innerResult*.[[Value]].

- b. If *return* is **undefined**, return `Completion(completion)`.
 - c. Set *innerResult* to `Call(return, iterator)`.
 - d. If *innerResult*.[[Type]] is normal, set *innerResult* to `Await(innerResult.[[Value]])`.
6. If *completion*.[[Type]] is throw, return `Completion(completion)`.
 7. If *innerResult*.[[Type]] is throw, return `Completion(innerResult)`.
 8. If `Type(innerResult.[[Value]])` is not Object, throw a **TypeError** exception.
 9. Return `Completion(completion)`.

7.4.8 CreateIterResultObject (*value*, *done*)

The abstract operation CreateIterResultObject takes arguments *value* and *done*. It creates an object that supports the IteratorResult interface. It performs the following steps when called:

1. Assert: `Type(done)` is Boolean.
2. Let *obj* be `! OrdinaryObjectCreate(%Object.prototype%)`.
3. Perform `! CreateDataPropertyOrThrow(obj, "value", value)`.
4. Perform `! CreateDataPropertyOrThrow(obj, "done", done)`.
5. Return *obj*.

7.4.9 CreateListIteratorRecord (*list*)

The abstract operation CreateListIteratorRecord takes argument *list*. It creates an Iterator (27.1.1.2) object record whose next method returns the successive elements of *list*. It performs the following steps when called:

1. Let *closure* be a new `Abstract Closure` with no parameters that captures *list* and performs the following steps when called:
 - a. For each element *E* of *list*, do
 - i. Perform `? Yield(E)`.
 - b. Return **undefined**.
2. Let *iterator* be `! CreateIteratorFromClosure(closure, empty, %IteratorPrototype%)`.
3. Return `Record { [[Iterator]]: iterator, [[NextMethod]]: %GeneratorFunction.prototype.next%, [[Done]]: false }`.

NOTE The list iterator object is never directly accessible to ECMAScript code.

7.4.10 IterableToList (*items* [, *method*])

The abstract operation IterableToList takes argument *items* and optional argument *method*. It performs the following steps when called:

1. If *method* is present, then
 - a. Let *iteratorRecord* be `? GetIterator(items, sync, method)`.
2. Else,
 - a. Let *iteratorRecord* be `? GetIterator(items, sync)`.
3. Let *values* be a new empty `List`.
4. Let *next* be **true**.
5. Repeat, while *next* is not **false**,

- a. Set *next* to ? `IteratorStep(iteratorRecord)`.
 - b. If *next* is not **false**, then
 - i. Let *nextValue* be ? `IteratorValue(next)`.
 - ii. Append *nextValue* to the end of the List *values*.
6. Return *values*.

8 Syntax-Directed Operations

In addition to those defined in this section, specialized syntax-directed operations are defined throughout this specification.

8.1 Scope Analysis

8.1.1 Static Semantics: BoundNames

NOTE **"*default*"** is used within this specification as a synthetic name for hoistable anonymous functions that are defined using export declarations.

BindingIdentifier : *Identifier*

1. Return a List whose sole element is the `StringValue` of *Identifier*.

BindingIdentifier : **yield**

1. Return a List whose sole element is **"yield"**.

BindingIdentifier : **await**

1. Return a List whose sole element is **"await"**.

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 Initializer*_{opt}

1. Return the `BoundNames` of *BindingIdentifier*.

LexicalBinding : *BindingPattern Initializer*

1. Return the `BoundNames` of *BindingPattern*.

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*_{opt}

1. Return the **BoundNames** of *BindingIdentifier*.

VariableDeclaration : *BindingPattern* *Initializer*

1. Return the **BoundNames** of *BindingPattern*.

ObjectBindingPattern : { }

1. Return a new empty **List**.

ObjectBindingPattern : { *BindingPropertyList* , *BindingRestProperty* }

1. Let *names* be **BoundNames** of *BindingPropertyList*.
2. Append to *names* the elements of **BoundNames** of *BindingRestProperty*.
3. Return *names*.

ArrayBindingPattern : [*Elision*_{opt}]

1. Return a new empty **List**.

ArrayBindingPattern : [*Elision*_{opt} *BindingRestElement*]

1. Return the **BoundNames** of *BindingRestElement*.

ArrayBindingPattern : [*BindingElementList* , *Elision*_{opt}]

1. Return the **BoundNames** of *BindingElementList*.

ArrayBindingPattern : [*BindingElementList* , *Elision*_{opt} *BindingRestElement*]

1. Let *names* be **BoundNames** of *BindingElementList*.
2. Append to *names* the elements of **BoundNames** of *BindingRestElement*.
3. Return *names*.

BindingPropertyList : *BindingPropertyList* , *BindingProperty*

1. Let *names* be **BoundNames** of *BindingPropertyList*.
2. Append to *names* the elements of **BoundNames** of *BindingProperty*.
3. Return *names*.

BindingElementList : *BindingElementList* , *BindingElisionElement*

1. Let *names* be **BoundNames** of *BindingElementList*.
2. Append to *names* the elements of **BoundNames** of *BindingElisionElement*.
3. Return *names*.

BindingElisionElement : *Elision*_{opt} *BindingElement*

1. Return **BoundNames** of *BindingElement*.

BindingProperty : *PropertyName* : *BindingElement*

1. Return the **BoundNames** of *BindingElement*.

SingleNameBinding : *BindingIdentifier* *Initializer*_{opt}

1. Return the **BoundNames** of *BindingIdentifier*.

BindingElement : *BindingPattern* *Initializer*_{opt}

1. Return the **BoundNames** of *BindingPattern*.

ForDeclaration : *LetOrConst* *ForBinding*

1. Return the **BoundNames** of *ForBinding*.

FunctionDeclaration : **function** *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return the **BoundNames** of *BindingIdentifier*.

FunctionDeclaration : **function** (*FormalParameters*) { *FunctionBody* }

1. Return « **"*default*"** ».

FormalParameters : [empty]

1. Return a new empty **List**.

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. Let *names* be **BoundNames** of *FormalParameterList*.
2. Append to *names* the **BoundNames** of *FunctionRestParameter*.
3. Return *names*.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. Let *names* be **BoundNames** of *FormalParameterList*.
2. Append to *names* the **BoundNames** of *FormalParameter*.
3. Return *names*.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be **CoveredFormalsList** of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the **BoundNames** of *formals*.

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *GeneratorBody* }

1. Return the **BoundNames** of *BindingIdentifier*.

GeneratorDeclaration : **function** * (*FormalParameters*) { *GeneratorBody* }

1. Return « **"*default*"** ».

AsyncGeneratorDeclaration : **async function** * *BindingIdentifier* (*FormalParameters*) { *AsyncGeneratorBody* }

1. Return the [BoundNames](#) of *BindingIdentifier*.

AsyncGeneratorDeclaration : **async function** * (*FormalParameters*) { *AsyncGeneratorBody* }

1. Return « **"*default*"** ».

ClassDeclaration : **class** *BindingIdentifier* *ClassTail*

1. Return the [BoundNames](#) of *BindingIdentifier*.

ClassDeclaration : **class** *ClassTail*

1. Return « **"*default*"** ».

AsyncFunctionDeclaration : **async function** *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

1. Return the [BoundNames](#) of *BindingIdentifier*.

AsyncFunctionDeclaration : **async function** (*FormalParameters*) { *AsyncFunctionBody* }

1. Return « **"*default*"** ».

CoverCallExpressionAndAsyncArrowHead : *MemberExpression* *Arguments*

1. Let *head* be [CoveredAsyncArrowHead](#) of *CoverCallExpressionAndAsyncArrowHead*.
2. Return the [BoundNames](#) of *head*.

ImportDeclaration : **import** *ImportClause* *FromClause* ;

1. Return the [BoundNames](#) of *ImportClause*.

ImportDeclaration : **import** *ModuleSpecifier* ;

1. Return a new empty [List](#).

ImportClause : *ImportedDefaultBinding* , *NameSpaceImport*

1. Let *names* be the [BoundNames](#) of *ImportedDefaultBinding*.
2. Append to *names* the elements of the [BoundNames](#) of *NameSpaceImport*.
3. Return *names*.

ImportClause : *ImportedDefaultBinding* , *NamedImports*

1. Let *names* be the [BoundNames](#) of *ImportedDefaultBinding*.
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*.

ExportDeclaration :

export *ExportFromClause* *FromClause* ;
export *NamedExports* ;

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** *HoistableDeclaration*

1. Let *declarationNames* be the **BoundNames** of *HoistableDeclaration*.
2. If *declarationNames* does not include the element **"*default*"**, append **"*default*"** to *declarationNames*.
3. Return *declarationNames*.

ExportDeclaration : **export default** *ClassDeclaration*

1. Let *declarationNames* be the **BoundNames** of *ClassDeclaration*.
2. If *declarationNames* does not include the element **"*default*"**, append **"*default*"** to *declarationNames*.
3. Return *declarationNames*.

ExportDeclaration : **export default** *AssignmentExpression* ;

1. Return « **"*default*"** ».

8.1.2 Static Semantics: DeclarationPart

HoistableDeclaration : *FunctionDeclaration*

1. Return *FunctionDeclaration*.

HoistableDeclaration : *GeneratorDeclaration*

1. Return *GeneratorDeclaration*.

HoistableDeclaration : *AsyncFunctionDeclaration*

1. Return *AsyncFunctionDeclaration*.

HoistableDeclaration : *AsyncGeneratorDeclaration*

1. Return *AsyncGeneratorDeclaration*.

Declaration : *ClassDeclaration*

1. Return *ClassDeclaration*.

Declaration : *LexicalDeclaration*

1. Return *LexicalDeclaration*.

8.1.3 Static Semantics: *IsConstantDeclaration*

LexicalDeclaration : *LetOrConst BindingList* ;

1. Return *IsConstantDeclaration* of *LetOrConst*.

LetOrConst : **let**

1. Return **false**.

LetOrConst : **const**

1. Return **true**.

FunctionDeclaration :

```
function BindingIdentifier ( FormalParameters ) { FunctionBody }  
function ( FormalParameters ) { FunctionBody }
```

GeneratorDeclaration :

```
function * BindingIdentifier ( FormalParameters ) { GeneratorBody }  
function * ( FormalParameters ) { GeneratorBody }
```

AsyncGeneratorDeclaration :

```
async function * BindingIdentifier ( FormalParameters ) { AsyncGeneratorBody }  
async function * ( FormalParameters ) { AsyncGeneratorBody }
```

AsyncFunctionDeclaration :

```
async function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }  
async function ( FormalParameters ) { AsyncFunctionBody }
```

1. Return **false**.

ClassDeclaration :

```
class BindingIdentifier ClassTail  
class ClassTail
```

1. Return **false**.

ExportDeclaration :

```
export ExportFromClause FromClause ;  
export NamedExports ;  
export default AssignmentExpression ;
```

1. Return **false**.

NOTE

It is not necessary to treat **export default** *AssignmentExpression* as a constant declaration because there is no syntax that permits assignment to the internal bound name used to reference a module's default object.

8.1.4 Static Semantics: LexicallyDeclaredNames

Block : { }

1. Return a new empty [List](#).

StatementList : *StatementList* *StatementListItem*

1. Let *names* be [LexicallyDeclaredNames](#) of *StatementList*.
2. Append to *names* the elements of the [LexicallyDeclaredNames](#) of *StatementListItem*.
3. Return *names*.

StatementListItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return [LexicallyDeclaredNames](#) of *LabelledStatement*.
2. Return a new empty [List](#).

StatementListItem : *Declaration*

1. Return the [BoundNames](#) of *Declaration*.

CaseBlock : { }

1. Return a new empty [List](#).

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. If the first *CaseClauses* is present, let *names* be the [LexicallyDeclaredNames](#) of the first *CaseClauses*.
2. Else, let *names* be a new empty [List](#).
3. Append to *names* the elements of the [LexicallyDeclaredNames](#) of *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Return the result of appending to *names* the elements of the [LexicallyDeclaredNames](#) of the second *CaseClauses*.

CaseClauses : *CaseClauses* *CaseClause*

1. Let *names* be [LexicallyDeclaredNames](#) of *CaseClauses*.
2. Append to *names* the elements of the [LexicallyDeclaredNames](#) of *CaseClause*.
3. Return *names*.

CaseClause : **case** *Expression* : *StatementList*_{opt}

1. If the *StatementList* is present, return the [LexicallyDeclaredNames](#) of *StatementList*.
2. Return a new empty [List](#).

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return the [LexicallyDeclaredNames](#) of *StatementList*.
2. Return a new empty [List](#).

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the [LexicallyDeclaredNames](#) of *LabelledItem*.

LabelledItem : *Statement*

1. Return a new empty [List](#).

LabelledItem : *FunctionDeclaration*

1. Return [BoundNames](#) of *FunctionDeclaration*.

FunctionStatementList : [empty]

1. Return a new empty [List](#).

FunctionStatementList : *StatementList*

1. Return [TopLevelLexicallyDeclaredNames](#) of *StatementList*.

ConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

AsyncConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

ScriptBody : *StatementList*

1. Return [TopLevelLexicallyDeclaredNames](#) of *StatementList*.

NOTE 1 At the top level of a *Script*, function declarations are treated like var declarations rather than like lexical declarations.

NOTE 2 The [LexicallyDeclaredNames](#) of a *Module* includes the names of all of its imported bindings.

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*, return a new empty [List](#).
2. Return the [BoundNames](#) of *ExportDeclaration*.

ModuleItem : *StatementListItem*

1. Return [LexicallyDeclaredNames](#) of *StatementListItem*.

NOTE 3 At the top level of a *Module*, function declarations are treated like lexical declarations rather than like var declarations.

8.1.5 Static Semantics: LexicallyScopedDeclarations

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* , return [LexicallyScopedDeclarations](#) of *LabelledStatement*.
2. Return a new empty [List](#).

StatementListItem : *Declaration*

1. Return a [List](#) whose sole element is [DeclarationPart](#) of *Declaration*.

CaseBlock : { }

1. Return a new empty [List](#).

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

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 *DefaultClause*.
4. If the second *CaseClauses* is not present, return *declarations*.
5. 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*_{opt}

1. If the *StatementList* is present, return the [LexicallyScopedDeclarations](#) of *StatementList*.
2. Return a new empty [List](#).

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return the [LexicallyScopedDeclarations](#) of *StatementList*.
2. Return a new empty [List](#).

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the [LexicallyScopedDeclarations](#) of *LabelledItem*.

LabelledItem : *Statement*

1. Return a new empty [List](#).

LabelledItem : *FunctionDeclaration*

1. Return a [List](#) whose sole element is *FunctionDeclaration*.

FunctionStatementList : [empty]

1. Return a new empty [List](#).

FunctionStatementList : *StatementList*

1. Return the [TopLevelLexicallyScopedDeclarations](#) of *StatementList*.

ConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

AsyncConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

ScriptBody : *StatementList*

1. Return [TopLevelLexicallyScopedDeclarations](#) of *StatementList*.

Module : [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. Return a new empty [List](#).

ExportDeclaration :

```
export ExportFromClause FromClause ;  
export NamedExports ;  
export VariableStatement
```

1. Return a new empty [List](#).

ExportDeclaration : **export** *Declaration*

1. Return a [List](#) whose sole element is [DeclarationPart](#) of *Declaration*.

ExportDeclaration : **export default** *HoistableDeclaration*

1. Return a [List](#) whose sole element is [DeclarationPart](#) of *HoistableDeclaration*.

ExportDeclaration : **export default** *ClassDeclaration*

1. Return a [List](#) whose sole element is *ClassDeclaration*.

ExportDeclaration : **export default** *AssignmentExpression* ;

1. Return a [List](#) whose sole element is this *ExportDeclaration*.

8.1.6 Static Semantics: VarDeclaredNames

Statement :

EmptyStatement
ExpressionStatement
ContinueStatement
BreakStatement
ReturnStatement
ThrowStatement
DebuggerStatement

1. Return a new empty [List](#).

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](#).

VariableStatement : **var** *VariableDeclarationList* ;

1. Return [BoundNames](#) of *VariableDeclarationList*.

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *names* be [VarDeclaredNames](#) of the first *Statement*.
2. Append to *names* the elements of the [VarDeclaredNames](#) of the second *Statement*.
3. Return *names*.

IfStatement : **if** (*Expression*) *Statement*

1. Return the [VarDeclaredNames](#) of *Statement*.

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

1. Return the [VarDeclaredNames](#) of *Statement*.

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

1. Return the [VarDeclaredNames](#) of *Statement*.

ForStatement : **for** (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return the [VarDeclaredNames](#) of *Statement*.

ForStatement : **for** (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Let *names* be **BoundNames** of *VariableDeclarationList*.
2. Append to *names* the elements of the **VarDeclaredNames** of *Statement*.
3. Return *names*.

ForStatement : **for** (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return the **VarDeclaredNames** of *Statement*.

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*
for (*ForDeclaration* **in** *Expression*) *Statement*
for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*
for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return the **VarDeclaredNames** of *Statement*.

ForInOfStatement :

for (**var** *ForBinding* **in** *Expression*) *Statement*
for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for await (**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*.

NOTE This section is extended by Annex [B.3.6](#).

WithStatement : **with** (*Expression*) *Statement*

1. Return the **VarDeclaredNames** of *Statement*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return the **VarDeclaredNames** of *CaseBlock*.

CaseBlock : { }

1. Return a new empty **List**.

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. If the first *CaseClauses* is present, let *names* be the **VarDeclaredNames** of the first *CaseClauses*.
2. Else, let *names* be a new empty **List**.
3. Append to *names* the elements of the **VarDeclaredNames** of *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Return the result of appending to *names* the elements of the **VarDeclaredNames** of the second *CaseClauses*.

CaseClauses : *CaseClauses* *CaseClause*

1. Let *names* be *VarDeclaredNames* of *CaseClauses*.
2. Append to *names* the elements of the *VarDeclaredNames* of *CaseClause*.
3. Return *names*.

CaseClause : **case** *Expression* : *StatementList*_{opt}

1. If the *StatementList* is present, return the *VarDeclaredNames* of *StatementList*.
2. Return a new empty *List*.

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return the *VarDeclaredNames* of *StatementList*.
2. Return a new empty *List*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the *VarDeclaredNames* of *LabelledItem*.

LabelledItem : *FunctionDeclaration*

1. Return a new empty *List*.

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*.

FunctionStatementList : [empty]

1. Return a new empty *List*.

FunctionStatementList : *StatementList*

1. Return *TopLevelVarDeclaredNames* of *StatementList*.

ConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

AsyncConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

ScriptBody : *StatementList*

1. Return [TopLevelVarDeclaredNames](#) of *StatementList*.

Module : [empty]

1. Return a new empty [List](#).

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 a new empty [List](#).

ModuleItem : *ExportDeclaration*

1. If *ExportDeclaration* is **export** *VariableStatement*, return [BoundNames](#) of *ExportDeclaration*.
2. Return a new empty [List](#).

8.1.7 Static Semantics: VarScopedDeclarations

Statement :

EmptyStatement

ExpressionStatement

ContinueStatement

BreakStatement

ReturnStatement

ThrowStatement

DebuggerStatement

1. Return a new empty [List](#).

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](#).

VariableDeclarationList : *VariableDeclaration*

1. Return a [List](#) whose sole element is *VariableDeclaration*.

VariableDeclarationList : *VariableDeclarationList* , *VariableDeclaration*

1. Let *declarations* be [VarScopedDeclarations](#) of *VariableDeclarationList*.
2. Append *VariableDeclaration* to *declarations*.
3. Return *declarations*.

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *declarations* be [VarScopedDeclarations](#) of the first *Statement*.
2. Append to *declarations* the elements of the [VarScopedDeclarations](#) of the second *Statement*.
3. Return *declarations*.

IfStatement : **if** (*Expression*) *Statement*

1. Return the [VarScopedDeclarations](#) of *Statement*.

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

1. Return the [VarScopedDeclarations](#) of *Statement*.

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

1. Return the [VarScopedDeclarations](#) of *Statement*.

ForStatement : **for** (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return the [VarScopedDeclarations](#) of *Statement*.

ForStatement : **for** (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Let *declarations* be [VarScopedDeclarations](#) of *VariableDeclarationList*.
2. Append to *declarations* the elements of the [VarScopedDeclarations](#) of *Statement*.
3. Return *declarations*.

ForStatement : **for** (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return the [VarScopedDeclarations](#) of *Statement*.

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*

for (*ForDeclaration* **in** *Expression*) *Statement*

for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return the [VarScopedDeclarations](#) of *Statement*.

ForInOfStatement :

```
for ( var ForBinding in Expression ) Statement
for ( var ForBinding of AssignmentExpression ) Statement
for await ( var ForBinding of AssignmentExpression ) Statement
```

1. Let *declarations* be a [List](#) whose sole element is *ForBinding*.
2. Append to *declarations* the elements of the [VarScopedDeclarations](#) of *Statement*.
3. Return *declarations*.

NOTE This section is extended by Annex [B.3.6](#).

WithStatement : **with** (*Expression*) *Statement*

1. Return the [VarScopedDeclarations](#) of *Statement*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return the [VarScopedDeclarations](#) of *CaseBlock*.

CaseBlock : { }

1. Return a new empty [List](#).

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

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 *DefaultClause*.
4. If the second *CaseClauses* is not present, return *declarations*.
5. 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* : *StatementList*_{opt}

1. If the *StatementList* is present, return the [VarScopedDeclarations](#) of *StatementList*.
2. Return a new empty [List](#).

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return the [VarScopedDeclarations](#) of *StatementList*.
2. Return a new empty [List](#).

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the [VarScopedDeclarations](#) of *LabelledItem*.

LabelledItem : *FunctionDeclaration*

1. Return a new empty [List](#).

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 : **catch** (*CatchParameter*) *Block*

1. Return the [VarScopedDeclarations](#) of *Block*.

FunctionStatementList : [empty]

1. Return a new empty [List](#).

FunctionStatementList : *StatementList*

1. Return the [TopLevelVarScopedDeclarations](#) of *StatementList*.

ConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

AsyncConciseBody : *ExpressionBody*

1. Return a new empty [List](#).

ScriptBody : *StatementList*

1. Return [TopLevelVarScopedDeclarations](#) of *StatementList*.

Module : [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*, return `VarScopedDeclarations` of *VariableStatement*.
2. Return a new empty `List`.

8.1.8 Static Semantics: TopLevelLexicallyDeclaredNames

StatementList : *StatementList* *StatementListItem*

1. Let *names* be `TopLevelLexicallyDeclaredNames` of *StatementList*.
2. Append to *names* the elements of the `TopLevelLexicallyDeclaredNames` of *StatementListItem*.
3. Return *names*.

StatementListItem : *Statement*

1. Return a new empty `List`.

StatementListItem : *Declaration*

1. If *Declaration* is *Declaration* : *HoistableDeclaration* , then
 - a. Return « ».
2. Return the `BoundNames` of *Declaration*.

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

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return a new empty `List`.

8.1.9 Static Semantics: TopLevelLexicallyScopedDeclarations

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* : *HoistableDeclaration* , then
 - a. Return « ».
2. Return a **List** whose sole element is *Declaration*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return a new empty **List**.

8.1.10 Static Semantics: TopLevelVarDeclaredNames

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* : *HoistableDeclaration* , then
 - a. Return the **BoundNames** of *HoistableDeclaration*.
2. Return a new empty **List**.

StatementListItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , 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.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the **TopLevelVarDeclaredNames** of *LabelledItem*.

LabelledItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return **TopLevelVarDeclaredNames** of *Statement*.
2. Return **VarDeclaredNames** of *Statement*.

LabelledItem : *FunctionDeclaration*

1. Return **BoundNames** of *FunctionDeclaration*.

8.1.11 Static Semantics: TopLevelVarScopedDeclarations

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* , return *TopLevelVarScopedDeclarations* of *Statement*.
2. Return *VarScopedDeclarations* of *Statement*.

StatementListItem : *Declaration*

1. If *Declaration* is *Declaration* : *HoistableDeclaration* , then
 - a. Let *declaration* be *DeclarationPart* of *HoistableDeclaration*.
 - b. Return « *declaration* ».
2. Return a new empty *List*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Return the *TopLevelVarScopedDeclarations* of *LabelledItem*.

LabelledItem : *Statement*

1. If *Statement* is *Statement* : *LabelledStatement* , return *TopLevelVarScopedDeclarations* of *Statement*.
2. Return *VarScopedDeclarations* of *Statement*.

LabelledItem : *FunctionDeclaration*

1. Return a *List* whose sole element is *FunctionDeclaration*.

8.2 Labels

8.2.1 Static Semantics: ContainsDuplicateLabels

With parameter *labelSet*.

Statement :

VariableStatement

EmptyStatement

ExpressionStatement

ContinueStatement

BreakStatement

ReturnStatement

ThrowStatement

DebuggerStatement

Block : { }

StatementListItem : *Declaration*

1. Return **false**.

StatementList : *StatementList* *StatementListItem*

1. Let *hasDuplicat*es be [ContainsDuplicateLabels](#) of *StatementList* with argument *labelSet*.
2. If *hasDuplicat*es is **true**, return **true**.
3. Return [ContainsDuplicateLabels](#) of *StatementListItem* with argument *labelSet*.

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *hasDuplicate* be [ContainsDuplicateLabels](#) of the first *Statement* with argument *labelSet*.
2. If *hasDuplicate* is **true**, return **true**.
3. Return [ContainsDuplicateLabels](#) of the second *Statement* with argument *labelSet*.

IfStatement : **if** (*Expression*) *Statement*

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

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

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

ForStatement :

for (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*
for (**var** *ForBinding* **in** *Expression*) *Statement*
for (*ForDeclaration* **in** *Expression*) *Statement*
for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*
for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for await (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

NOTE This section is extended by Annex B.3.6.

WithStatement : **with** (*Expression*) *Statement*

1. Return [ContainsDuplicateLabels](#) of *Statement* with argument *labelSet*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return `ContainsDuplicateLabels` of `CaseBlock` with argument `labelSet`.

`CaseBlock` : { }

1. Return **false**.

`CaseBlock` : { `CaseClauses`_{opt} `DefaultClause` `CaseClauses`_{opt} }

1. If the first `CaseClauses` is present, then
 - a. Let `hasDuplicatess` be `ContainsDuplicateLabels` of the first `CaseClauses` with argument `labelSet`.
 - b. If `hasDuplicatess` is **true**, return **true**.
2. Let `hasDuplicatess` be `ContainsDuplicateLabels` of `DefaultClause` with argument `labelSet`.
3. If `hasDuplicatess` is **true**, return **true**.
4. If the second `CaseClauses` is not present, return **false**.
5. Return `ContainsDuplicateLabels` of the second `CaseClauses` with argument `labelSet`.

`CaseClauses` : `CaseClauses` `CaseClause`

1. Let `hasDuplicatess` be `ContainsDuplicateLabels` of `CaseClauses` with argument `labelSet`.
2. If `hasDuplicatess` is **true**, return **true**.
3. Return `ContainsDuplicateLabels` of `CaseClause` with argument `labelSet`.

`CaseClause` : **case** `Expression` : `StatementList`_{opt}

1. If the `StatementList` is present, return `ContainsDuplicateLabels` of `StatementList` with argument `labelSet`.
2. Return **false**.

`DefaultClause` : **default** : `StatementList`_{opt}

1. If the `StatementList` is present, return `ContainsDuplicateLabels` of `StatementList` with argument `labelSet`.
2. Return **false**.

`LabelledStatement` : `LabelIdentifier` : `LabelledItem`

1. Let `label` be the `StringValue` of `LabelIdentifier`.
2. If `label` is an element of `labelSet`, return **true**.
3. Let `newLabelSet` be a copy of `labelSet` with `label` appended.
4. Return `ContainsDuplicateLabels` of `LabelledItem` with argument `newLabelSet`.

`LabelledItem` : `FunctionDeclaration`

1. Return **false**.

`TryStatement` : **try** `Block` `Catch`

1. Let `hasDuplicatess` be `ContainsDuplicateLabels` of `Block` with argument `labelSet`.
2. If `hasDuplicatess` is **true**, return **true**.
3. Return `ContainsDuplicateLabels` of `Catch` with argument `labelSet`.

`TryStatement` : **try** `Block` `Finally`

1. Let `hasDuplicatess` be `ContainsDuplicateLabels` of `Block` with argument `labelSet`.
2. If `hasDuplicatess` is **true**, return **true**.

3. Return `ContainsDuplicateLabels` of *Finally* with argument *labelSet*.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *hasDuplicates* be `ContainsDuplicateLabels` of *Block* with argument *labelSet*.
2. If *hasDuplicates* is **true**, return **true**.
3. Let *hasDuplicates* be `ContainsDuplicateLabels` of *Catch* with argument *labelSet*.
4. If *hasDuplicates* is **true**, return **true**.
5. Return `ContainsDuplicateLabels` of *Finally* with argument *labelSet*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return `ContainsDuplicateLabels` of *Block* with argument *labelSet*.

FunctionStatementList : [empty]

1. Return **false**.

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *hasDuplicates* be `ContainsDuplicateLabels` of *ModuleItemList* with argument *labelSet*.
2. If *hasDuplicates* is **true**, return **true**.
3. Return `ContainsDuplicateLabels` of *ModuleItem* with argument *labelSet*.

ModuleItem :

ImportDeclaration

ExportDeclaration

1. Return **false**.

8.2.2 Static Semantics: `ContainsUndefinedBreakTarget`

With parameter *labelSet*.

Statement :

VariableStatement

EmptyStatement

ExpressionStatement

ContinueStatement

ReturnStatement

ThrowStatement

DebuggerStatement

Block : { }

StatementListItem : *Declaration*

1. Return **false**.

StatementList : *StatementList* *StatementListItem*

1. Let *hasUndefinedLabels* be `ContainsUndefinedBreakTarget` of *StatementList* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.

3. Return `ContainsUndefinedBreakTarget` of `StatementListItem` with argument `labelSet`.

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let `hasUndefinedLabels` be `ContainsUndefinedBreakTarget` of the first `Statement` with argument `labelSet`.
2. If `hasUndefinedLabels` is **true**, return **true**.
3. Return `ContainsUndefinedBreakTarget` of the second `Statement` with argument `labelSet`.

IfStatement : **if** (*Expression*) *Statement*

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

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

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

ForStatement :

for (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

for (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

for (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*

for (**var** *ForBinding* **in** *Expression*) *Statement*

for (*ForDeclaration* **in** *Expression*) *Statement*

for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

for await (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

NOTE This section is extended by Annex B.3.6.

BreakStatement : **break** ;

1. Return **false**.

BreakStatement : **break** *LabelIdentifier* ;

1. If the `StringValue` of `LabelIdentifier` is not an element of `labelSet`, return **true**.
2. Return **false**.

WithStatement : **with** (*Expression*) *Statement*

1. Return **ContainsUndefinedBreakTarget** of *Statement* with argument *labelSet*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return **ContainsUndefinedBreakTarget** of *CaseBlock* with argument *labelSet*.

CaseBlock : { }

1. Return **false**.

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. If the first *CaseClauses* is present, then
 - a. Let *hasUndefinedLabels* be **ContainsUndefinedBreakTarget** of the first *CaseClauses* with argument *labelSet*.
 - b. If *hasUndefinedLabels* is **true**, return **true**.
2. Let *hasUndefinedLabels* be **ContainsUndefinedBreakTarget** of *DefaultClause* with argument *labelSet*.
3. If *hasUndefinedLabels* is **true**, return **true**.
4. If the second *CaseClauses* is not present, return **false**.
5. Return **ContainsUndefinedBreakTarget** of the second *CaseClauses* with argument *labelSet*.

CaseClauses : *CaseClauses* *CaseClause*

1. Let *hasUndefinedLabels* be **ContainsUndefinedBreakTarget** of *CaseClauses* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return **ContainsUndefinedBreakTarget** of *CaseClause* with argument *labelSet*.

CaseClause : **case** *Expression* : *StatementList*_{opt}

1. If the *StatementList* is present, return **ContainsUndefinedBreakTarget** of *StatementList* with argument *labelSet*.
2. Return **false**.

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return **ContainsUndefinedBreakTarget** of *StatementList* with argument *labelSet*.
2. Return **false**.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the **StringValue** of *LabelIdentifier*.
2. Let *newLabelSet* be a copy of *labelSet* with *label* appended.
3. Return **ContainsUndefinedBreakTarget** of *LabelledItem* with argument *newLabelSet*.

LabelledItem : *FunctionDeclaration*

1. Return **false**.

TryStatement : **try** *Block* *Catch*

1. Let *hasUndefinedLabels* be **ContainsUndefinedBreakTarget** of *Block* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return **ContainsUndefinedBreakTarget** of *Catch* with argument *labelSet*.

TryStatement : **try** *Block* *Finally*

1. Let *hasUndefinedLabels* be *ContainsUndefinedBreakTarget* of *Block* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedBreakTarget* of *Finally* with argument *labelSet*.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *hasUndefinedLabels* be *ContainsUndefinedBreakTarget* of *Block* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Let *hasUndefinedLabels* be *ContainsUndefinedBreakTarget* of *Catch* with argument *labelSet*.
4. If *hasUndefinedLabels* is **true**, return **true**.
5. Return *ContainsUndefinedBreakTarget* of *Finally* with argument *labelSet*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return *ContainsUndefinedBreakTarget* of *Block* with argument *labelSet*.

FunctionStatementList : [empty]

1. Return **false**.

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *hasUndefinedLabels* be *ContainsUndefinedBreakTarget* of *ModuleItemList* with argument *labelSet*.
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedBreakTarget* of *ModuleItem* with argument *labelSet*.

ModuleItem :

ImportDeclaration

ExportDeclaration

1. Return **false**.

8.2.3 Static Semantics: *ContainsUndefinedContinueTarget*

With parameters *iterationSet* and *labelSet*.

Statement :

VariableStatement

EmptyStatement

ExpressionStatement

BreakStatement

ReturnStatement

ThrowStatement

DebuggerStatement

Block : { }

StatementListItem : *Declaration*

1. Return **false**.

BreakableStatement : *IterationStatement*

1. Let *newIterationSet* be a copy of *iterationSet* with all the elements of *labelSet* appended.
2. Return **ContainsUndefinedContinueTarget** of *IterationStatement* with arguments *newIterationSet* and « ».

StatementList : *StatementList* *StatementListItem*

1. Let *hasUndefinedLabels* be **ContainsUndefinedContinueTarget** of *StatementList* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return **ContainsUndefinedContinueTarget** of *StatementListItem* with arguments *iterationSet* and « ».

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *hasUndefinedLabels* be **ContainsUndefinedContinueTarget** of the first *Statement* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return **ContainsUndefinedContinueTarget** of the second *Statement* with arguments *iterationSet* and « ».

IfStatement : **if** (*Expression*) *Statement*

1. Return **ContainsUndefinedContinueTarget** of *Statement* with arguments *iterationSet* and « ».

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

1. Return **ContainsUndefinedContinueTarget** of *Statement* with arguments *iterationSet* and « ».

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

1. Return **ContainsUndefinedContinueTarget** of *Statement* with arguments *iterationSet* and « ».

ForStatement :

for (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*
for (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Return **ContainsUndefinedContinueTarget** of *Statement* with arguments *iterationSet* and « ».

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*
for (**var** *ForBinding* **in** *Expression*) *Statement*
for (*ForDeclaration* **in** *Expression*) *Statement*
for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*
for await (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*
for await (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*
for await (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

1. Return **ContainsUndefinedContinueTarget** of *Statement* with arguments *iterationSet* and « ».

ContinueStatement : **continue** ;

1. Return **false**.

ContinueStatement : **continue** *LabelIdentifier* ;

1. If the *StringValue* of *LabelIdentifier* is not an element of *iterationSet*, return **true**.
2. Return **false**.

WithStatement : **with** (*Expression*) *Statement*

1. Return *ContainsUndefinedContinueTarget* of *Statement* with arguments *iterationSet* and « ».

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return *ContainsUndefinedContinueTarget* of *CaseBlock* with arguments *iterationSet* and « ».

CaseBlock : { }

1. Return **false**.

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. If the first *CaseClauses* is present, then
 - a. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of the first *CaseClauses* with arguments *iterationSet* and « ».
 - b. If *hasUndefinedLabels* is **true**, return **true**.
2. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *DefaultClause* with arguments *iterationSet* and « ».
3. If *hasUndefinedLabels* is **true**, return **true**.
4. If the second *CaseClauses* is not present, return **false**.
5. Return *ContainsUndefinedContinueTarget* of the second *CaseClauses* with arguments *iterationSet* and « ».

CaseClauses : *CaseClauses* *CaseClause*

1. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *CaseClauses* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedContinueTarget* of *CaseClause* with arguments *iterationSet* and « ».

CaseClause : **case** *Expression* : *StatementList*_{opt}

1. If the *StatementList* is present, return *ContainsUndefinedContinueTarget* of *StatementList* with arguments *iterationSet* and « ».
2. Return **false**.

DefaultClause : **default** : *StatementList*_{opt}

1. If the *StatementList* is present, return *ContainsUndefinedContinueTarget* of *StatementList* with arguments *iterationSet* and « ».
2. Return **false**.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the *StringValue* of *LabelIdentifier*.
2. Let *newLabelSet* be a copy of *labelSet* with *label* appended.
3. Return *ContainsUndefinedContinueTarget* of *LabelledItem* with arguments *iterationSet* and *newLabelSet*.

LabelledItem : *FunctionDeclaration*

1. Return **false**.

TryStatement : **try** *Block* *Catch*

1. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *Block* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedContinueTarget* of *Catch* with arguments *iterationSet* and « ».

TryStatement : **try** *Block* *Finally*

1. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *Block* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedContinueTarget* of *Finally* with arguments *iterationSet* and « ».

TryStatement : **try** *Block* *Catch* *Finally*

1. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *Block* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *Catch* with arguments *iterationSet* and « ».
4. If *hasUndefinedLabels* is **true**, return **true**.
5. Return *ContainsUndefinedContinueTarget* of *Finally* with arguments *iterationSet* and « ».

Catch : **catch** (*CatchParameter*) *Block*

1. Return *ContainsUndefinedContinueTarget* of *Block* with arguments *iterationSet* and « ».

FunctionStatementList : [empty]

1. Return **false**.

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *hasUndefinedLabels* be *ContainsUndefinedContinueTarget* of *ModuleItemList* with arguments *iterationSet* and « ».
2. If *hasUndefinedLabels* is **true**, return **true**.
3. Return *ContainsUndefinedContinueTarget* of *ModuleItem* with arguments *iterationSet* and « ».

ModuleItem :

ImportDeclaration

ExportDeclaration

1. Return **false**.

8.3 Function Name Inference

8.3.1 Static Semantics: HasName

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoveredParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. If *IsFunctionDefinition* of *expr* is **false**, return **false**.
3. Return *HasName* of *expr*.

FunctionExpression :

function (*FormalParameters*) { *FunctionBody* }

GeneratorExpression :

function * (*FormalParameters*) { *GeneratorBody* }

AsyncGeneratorExpression :

async function * (*FormalParameters*) { *AsyncGeneratorBody* }

AsyncFunctionExpression :

async function (*FormalParameters*) { *AsyncFunctionBody* }

ArrowFunction :

ArrowParameters => *ConciseBody*

AsyncArrowFunction :

async *AsyncArrowBindingIdentifier* => *AsyncConciseBody*
CoverCallExpressionAndAsyncArrowHead => *AsyncConciseBody*

ClassExpression : **class** *ClassTail*

1. Return **false**.

FunctionExpression :

function *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

GeneratorExpression :

function * *BindingIdentifier* (*FormalParameters*) { *GeneratorBody* }

AsyncGeneratorExpression :

async function * *BindingIdentifier* (*FormalParameters*) { *AsyncGeneratorBody* }

AsyncFunctionExpression :

async function *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

ClassExpression : **class** *BindingIdentifier* *ClassTail*

1. Return **true**.

8.3.2 Static Semantics: IsFunctionDefinition

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

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

PrimaryExpression :

this

IdentifierReference

Literal

ArrayLiteral

ObjectLiteral

RegularExpressionLiteral

TemplateLiteral

MemberExpression :

MemberExpression [*Expression*]

MemberExpression . *IdentifierName*

MemberExpression *TemplateLiteral*

SuperProperty

MetaProperty

new *MemberExpression* *Arguments*

NewExpression :

new *NewExpression*

LeftHandSideExpression :

CallExpression

OptionalExpression

UpdateExpression :

LeftHandSideExpression ++

LeftHandSideExpression --

++ *UnaryExpression*

-- *UnaryExpression*

UnaryExpression :

delete *UnaryExpression*

void *UnaryExpression*

typeof *UnaryExpression*

+ *UnaryExpression*

- *UnaryExpression*

~ *UnaryExpression*

! *UnaryExpression*

AwaitExpression

ExponentiationExpression :

UpdateExpression ** *ExponentiationExpression*

MultiplicativeExpression :

MultiplicativeExpression *MultiplicativeOperator* *ExponentiationExpression*

AdditiveExpression :

AdditiveExpression + *MultiplicativeExpression*

AdditiveExpression - *MultiplicativeExpression*

ShiftExpression :

ShiftExpression << *AdditiveExpression*

ShiftExpression >> *AdditiveExpression*

ShiftExpression >>> *AdditiveExpression*

RelationalExpression :

RelationalExpression < *ShiftExpression*

RelationalExpression > *ShiftExpression*

RelationalExpression <= *ShiftExpression*

RelationalExpression >= *ShiftExpression*

RelationalExpression **instanceof** *ShiftExpression*
RelationalExpression **in** *ShiftExpression*
EqualityExpression :
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*
BitwiseANDExpression :
BitwiseANDExpression **&** *EqualityExpression*
BitwiseXORExpression :
BitwiseXORExpression **^** *BitwiseANDExpression*
BitwiseORExpression :
BitwiseORExpression **|** *BitwiseXORExpression*
LogicalANDExpression :
LogicalANDExpression **&&** *BitwiseORExpression*
LogicalORExpression :
LogicalORExpression **||** *LogicalANDExpression*
CoalesceExpression :
CoalesceExpressionHead **??** *BitwiseORExpression*
ConditionalExpression :
ShortCircuitExpression **?** *AssignmentExpression* **:** *AssignmentExpression*
AssignmentExpression :
YieldExpression
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression *AssignmentOperator* *AssignmentExpression*
LeftHandSideExpression **&&=** *AssignmentExpression*
LeftHandSideExpression **||=** *AssignmentExpression*
LeftHandSideExpression **??=** *AssignmentExpression*
Expression :
Expression **,** *AssignmentExpression*

1. Return false.

AssignmentExpression :
ArrowFunction
AsyncArrowFunction
FunctionExpression :
function *BindingIdentifier*_{opt} (*FormalParameters*) { *FunctionBody* }
GeneratorExpression :
function * *BindingIdentifier*_{opt} (*FormalParameters*) { *GeneratorBody* }
AsyncGeneratorExpression :
async function * *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncGeneratorBody* }
AsyncFunctionExpression :
async function *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncFunctionBody* }
ClassExpression : **class** *BindingIdentifier*_{opt} *ClassTail*

1. Return **true**.

8.3.3 Static Semantics: IsAnonymousFunctionDefinition (*expr*)

The abstract operation IsAnonymousFunctionDefinition takes argument *expr* (a [Parse Node](#) for *AssignmentExpression* or a [Parse Node](#) for *Initializer*). It determines if its argument is a function definition that does not bind a name. It performs the following steps when called:

1. If [IsFunctionDefinition](#) of *expr* is **false**, return **false**.
2. Let *hasName* be [HasName](#) of *expr*.
3. If *hasName* is **true**, return **false**.
4. Return **true**.

8.3.4 Static Semantics: IsIdentifierRef

PrimaryExpression : *IdentifierReference*

1. Return **true**.

PrimaryExpression :

this

Literal

ArrayLiteral

ObjectLiteral

FunctionExpression

ClassExpression

GeneratorExpression

AsyncFunctionExpression

AsyncGeneratorExpression

RegularExpressionLiteral

TemplateLiteral

CoverParenthesizedExpressionAndArrowParameterList

MemberExpression :

MemberExpression [*Expression*]

MemberExpression . *IdentifierName*

MemberExpression *TemplateLiteral*

SuperProperty

MetaProperty

new *MemberExpression* *Arguments*

NewExpression :

new *NewExpression*

LeftHandSideExpression :

CallExpression

OptionalExpression

1. Return **false**.

8.3.5 Runtime Semantics: NamedEvaluation

With parameter *name*.

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoveredParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the result of performing *NamedEvaluation* for *expr* with argument *name*.

ParenthesizedExpression : (*Expression*)

1. **Assert: *IsAnonymousFunctionDefinition(Expression)* is true.**
2. Return the result of performing *NamedEvaluation* for *Expression* with argument *name*.

FunctionExpression : **function** (*FormalParameters*) { *FunctionBody* }

1. Return *InstantiateOrdinaryFunctionExpression* of *FunctionExpression* with argument *name*.

GeneratorExpression : **function** * (*FormalParameters*) { *GeneratorBody* }

1. Return *InstantiateGeneratorFunctionExpression* of *GeneratorExpression* with argument *name*.

AsyncGeneratorExpression : **async function** * (*FormalParameters*) { *AsyncGeneratorBody* }

1. Return *InstantiateAsyncGeneratorFunctionExpression* of *AsyncGeneratorExpression* with argument *name*.

AsyncFunctionExpression : **async function** (*FormalParameters*) { *AsyncFunctionBody* }

1. Return *InstantiateAsyncFunctionExpression* of *AsyncFunctionExpression* with argument *name*.

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. Return *InstantiateArrowFunctionExpression* of *ArrowFunction* with argument *name*.

AsyncArrowFunction :

async *AsyncArrowBindingIdentifier* => *AsyncConciseBody*
CoverCallExpressionAndAsyncArrowHead => *AsyncConciseBody*

1. Return *InstantiateAsyncArrowFunctionExpression* of *AsyncArrowFunction* with argument *name*.

ClassExpression : **class** *ClassTail*

1. Let *value* be the result of *ClassDefinitionEvaluation* of *ClassTail* with arguments **undefined** and *name*.
2. *ReturnIfAbrupt(value)*.
3. Set *value*.[[SourceText]] to the source text matched by *ClassExpression*.
4. Return *value*.

8.4 Contains

8.4.1 Static Semantics: Contains

With parameter *symbol*.

Every grammar production alternative in this specification which is not listed below implicitly has the following default definition of Contains:

1. For each child node *child* of this **Parse Node**, do
 - a. If *child* is an instance of *symbol*, return **true**.
 - b. If *child* is an instance of a nonterminal, then
 - i. Let *contained* be the result of *child* **Contains** *symbol*.
 - ii. If *contained* is **true**, return **true**.
2. Return **false**.

FunctionDeclaration :

```
function BindingIdentifier ( FormalParameters ) { FunctionBody }  
function ( FormalParameters ) { FunctionBody }
```

FunctionExpression :

```
function BindingIdentifieropt ( FormalParameters ) { FunctionBody }
```

GeneratorDeclaration :

```
function * BindingIdentifier ( FormalParameters ) { GeneratorBody }  
function * ( FormalParameters ) { GeneratorBody }
```

GeneratorExpression :

```
function * BindingIdentifieropt ( FormalParameters ) { GeneratorBody }
```

AsyncGeneratorDeclaration :

```
async function * BindingIdentifier ( FormalParameters ) { AsyncGeneratorBody }  
async function * ( FormalParameters ) { AsyncGeneratorBody }
```

AsyncGeneratorExpression :

```
async function * BindingIdentifieropt ( FormalParameters ) { AsyncGeneratorBody }
```

AsyncFunctionDeclaration :

```
async function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }  
async function ( FormalParameters ) { AsyncFunctionBody }
```

AsyncFunctionExpression :

```
async function BindingIdentifieropt ( FormalParameters ) { AsyncFunctionBody }
```

1. Return **false**.

NOTE 1 Static semantic rules that depend upon substructure generally do not look into function definitions.

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**, return **true**.
5. Return the result of **ComputedPropertyContains** for *ClassBody* with argument *symbol*.

NOTE 2 Static semantic rules that depend upon substructure generally do not look into class bodies except for *PropertyNames*.

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. If *symbol* is not one of *NewTarget*, *SuperProperty*, *SuperCall*, **super** or **this**, return **false**.
2. If *ArrowParameters* **Contains** *symbol* is **true**, return **true**.
3. Return *ConciseBody* **Contains** *symbol*.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be **CoveredFormalsList** of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return *formals* **Contains** *symbol*.

AsyncArrowFunction : **async** *AsyncArrowBindingIdentifier* => *AsyncConciseBody*

1. If *symbol* is not one of *NewTarget*, *SuperProperty*, *SuperCall*, **super**, or **this**, return **false**.
2. Return *AsyncConciseBody* **Contains** *symbol*.

AsyncArrowFunction : *CoverCallExpressionAndAsyncArrowHead* => *AsyncConciseBody*

1. If *symbol* is not one of *NewTarget*, *SuperProperty*, *SuperCall*, **super**, or **this**, return **false**.
2. Let *head* be **CoveredAsyncArrowHead** of *CoverCallExpressionAndAsyncArrowHead*.
3. If *head* **Contains** *symbol* is **true**, return **true**.
4. Return *AsyncConciseBody* **Contains** *symbol*.

NOTE 3 **Contains** is used to detect **new.target**, **this**, and **super** usage within an *ArrowFunction* or *AsyncArrowFunction*.

PropertyDefinition : *MethodDefinition*

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

LiteralPropertyName : *IdentifierName*

1. Return **false**.

MemberExpression : *MemberExpression* . *IdentifierName*

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

SuperProperty : **super** . *IdentifierName*

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

CallExpression : *CallExpression* . *IdentifierName*

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

OptionalChain : ?. *IdentifierName*

1. Return **false**.

OptionalChain : *OptionalChain* . *IdentifierName*

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

8.4.2 Static Semantics: **ComputedPropertyContains**

With parameter *symbol*.

PropertyName : *LiteralPropertyName*

1. Return **false**.

PropertyName : *ComputedPropertyName*

1. Return the result of *ComputedPropertyName* **Contains** *symbol*.

MethodDefinition :

```
PropertyName ( UniqueFormalParameters ) { FunctionBody }  
get PropertyName ( ) { FunctionBody }  
set PropertyName ( PropertySetParameterList ) { FunctionBody }
```

1. Return the result of **ComputedPropertyContains** for *PropertyName* with argument *symbol*.

GeneratorMethod : * *PropertyName* (*UniqueFormalParameters*) { *GeneratorBody* }

1. Return the result of **ComputedPropertyContains** for *PropertyName* with argument *symbol*.

AsyncGeneratorMethod : **async** * *PropertyName* (*UniqueFormalParameters*) { *AsyncGeneratorBody* }

1. Return the result of **ComputedPropertyContains** for *PropertyName* with argument *symbol*.

ClassElementList : *ClassElementList* *ClassElement*

1. Let *inList* be **ComputedPropertyContains** of *ClassElementList* with argument *symbol*.
2. If *inList* is **true**, return **true**.
3. Return the result of **ComputedPropertyContains** for *ClassElement* with argument *symbol*.

ClassElement : ;

1. Return **false**.

AsyncMethod : **async** *PropertyName* (*UniqueFormalParameters*) { *AsyncFunctionBody* }

1. Return the result of **ComputedPropertyContains** for *PropertyName* with argument *symbol*.

8.5 Miscellaneous

These operations are used in multiple places throughout the specification.

8.5.1 Runtime Semantics: InstantiateFunctionObject

With parameter *scope*.

FunctionDeclaration :

```
function BindingIdentifier ( FormalParameters ) { FunctionBody }  
function ( FormalParameters ) { FunctionBody }
```

1. Return ? [InstantiateOrdinaryFunctionObject](#) of *FunctionDeclaration* with argument *scope*.

GeneratorDeclaration :

```
function * BindingIdentifier ( FormalParameters ) { GeneratorBody }  
function * ( FormalParameters ) { GeneratorBody }
```

1. Return ? [InstantiateGeneratorFunctionObject](#) of *GeneratorDeclaration* with argument *scope*.

AsyncGeneratorDeclaration :

```
async function * BindingIdentifier ( FormalParameters ) { AsyncGeneratorBody }  
async function * ( FormalParameters ) { AsyncGeneratorBody }
```

1. Return ? [InstantiateAsyncGeneratorFunctionObject](#) of *AsyncGeneratorDeclaration* with argument *scope*.

AsyncFunctionDeclaration :

```
async function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }  
async function ( FormalParameters ) { AsyncFunctionBody }
```

1. Return ? [InstantiateAsyncFunctionObject](#) of *AsyncFunctionDeclaration* with argument *scope*.

8.5.2 Runtime Semantics: BindingInitialization

With parameters *value* and *environment*.

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 10.2.10). 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 : **yield**

1. Return ? [InitializeBoundName](#)("yield", *value*, *environment*).

BindingIdentifier : **await**

1. Return ? [InitializeBoundName](#)("await", *value*, *environment*).

BindingPattern : *ObjectBindingPattern*

1. Perform ? [RequireObjectCoercible](#)(*value*).
2. Return the result of performing [BindingInitialization](#) for *ObjectBindingPattern* using *value* and *environment* as arguments.

BindingPattern : *ArrayBindingPattern*

1. Let *iteratorRecord* be ? [GetIterator](#)(*value*).
2. Let *result* be [IteratorBindingInitialization](#) of *ArrayBindingPattern* with arguments *iteratorRecord* and *environment*.
3. If *iteratorRecord*.[[Done]] is **false**, return ? [IteratorClose](#)(*iteratorRecord*, *result*).
4. Return *result*.

ObjectBindingPattern : { }

1. Return [NormalCompletion](#)(empty).

ObjectBindingPattern :

{ *BindingPropertyList* }

{ *BindingPropertyList* , }

1. Perform ? [PropertyBindingInitialization](#) for *BindingPropertyList* using *value* and *environment* as the arguments.
2. Return [NormalCompletion](#)(empty).

ObjectBindingPattern : { *BindingRestProperty* }

1. Let *excludedNames* be a new empty [List](#).
2. Return the result of performing [RestBindingInitialization](#) of *BindingRestProperty* with *value*, *environment*, and *excludedNames* as the arguments.

ObjectBindingPattern : { *BindingPropertyList* , *BindingRestProperty* }

1. Let *excludedNames* be ? [PropertyBindingInitialization](#) of *BindingPropertyList* with arguments *value* and *environment*.
2. Return the result of performing [RestBindingInitialization](#) of *BindingRestProperty* with arguments *value*, *environment*, and *excludedNames*.

8.5.2.1 InitializeBoundName (*name*, *value*, *environment*)

The abstract operation [InitializeBoundName](#) takes arguments *name*, *value*, and *environment*. It performs the following steps when called:

1. [Assert](#): [Type](#)(*name*) is String.
2. If *environment* is not **undefined**, then
 - a. Perform *environment*.[InitializeBinding](#)(*name*, *value*).
 - b. Return [NormalCompletion](#)(**undefined**).
3. Else,
 - a. Let *lhs* be [ResolveBinding](#)(*name*).
 - b. Return ? [PutValue](#)(*lhs*, *value*).

8.5.3 Runtime Semantics: IteratorBindingInitialization

With parameters *iteratorRecord* 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 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 *iteratorRecord* as the argument.

ArrayBindingPattern : [*Elision*_{opt} *BindingRestElement*]

1. If *Elision* is present, then
 - a. Perform ? **IteratorDestructuringAssignmentEvaluation** of *Elision* with *iteratorRecord* as the argument.
2. Return the result of performing **IteratorBindingInitialization** for *BindingRestElement* with *iteratorRecord* and *environment* as arguments.

ArrayBindingPattern : [*BindingElementList* , *Elision*]

1. Perform ? **IteratorBindingInitialization** for *BindingElementList* with *iteratorRecord* and *environment* as arguments.
2. Return the result of performing **IteratorDestructuringAssignmentEvaluation** of *Elision* with *iteratorRecord* as the argument.

ArrayBindingPattern : [*BindingElementList* , *Elision*_{opt} *BindingRestElement*]

1. Perform ? **IteratorBindingInitialization** for *BindingElementList* with *iteratorRecord* and *environment* as arguments.
2. If *Elision* is present, then
 - a. Perform ? **IteratorDestructuringAssignmentEvaluation** of *Elision* with *iteratorRecord* as the argument.
3. Return the result of performing **IteratorBindingInitialization** for *BindingRestElement* with *iteratorRecord* and *environment* as arguments.

BindingElementList : *BindingElementList* , *BindingElisionElement*

1. Perform ? **IteratorBindingInitialization** for *BindingElementList* with *iteratorRecord* and *environment* as arguments.
2. Return the result of performing **IteratorBindingInitialization** for *BindingElisionElement* using *iteratorRecord* and *environment* as arguments.

BindingElisionElement : *Elision* *BindingElement*

1. Perform ? **IteratorDestructuringAssignmentEvaluation** of *Elision* with *iteratorRecord* as the argument.
2. Return the result of performing **IteratorBindingInitialization** of *BindingElement* with *iteratorRecord* and *environment* as the arguments.

SingleNameBinding : *BindingIdentifier* *Initializer*_{opt}

1. Let *bindingId* be **StringValue** of *BindingIdentifier*.
2. Let *lhs* be ? **ResolveBinding**(*bindingId*, *environment*).
3. If *iteratorRecord*.[[Done]] is **false**, then

- a. Let *next* be `IteratorStep(iteratorRecord)`.
- b. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
- c. `ReturnIfAbrupt(next)`.
- d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
- e. Else,
 - i. Let *v* be `IteratorValue(next)`.
 - ii. If *v* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - iii. `ReturnIfAbrupt(v)`.
4. If *iteratorRecord*.[[Done]] is **true**, let *v* be **undefined**.
5. If *Initializer* is present and *v* is **undefined**, then
 - a. If `IsAnonymousFunctionDefinition(Initializer)` is **true**, then
 - i. Set *v* to the result of performing `NamedEvaluation` for *Initializer* with argument *bindingId*.
 - b. Else,
 - i. Let *defaultValue* be the result of evaluating *Initializer*.
 - ii. Set *v* to ? `GetValue(defaultValue)`.
6. If *environment* is **undefined**, return ? `PutValue(lhs, v)`.
7. Return `InitializeReferencedBinding(lhs, v)`.

BindingElement : *BindingPattern* *Initializer*_{opt}

1. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be `IteratorStep(iteratorRecord)`.
 - b. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - c. `ReturnIfAbrupt(next)`.
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
 - e. Else,
 - i. Let *v* be `IteratorValue(next)`.
 - ii. If *v* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - iii. `ReturnIfAbrupt(v)`.
2. If *iteratorRecord*.[[Done]] is **true**, let *v* be **undefined**.
3. If *Initializer* is present and *v* is **undefined**, then
 - a. Let *defaultValue* be the result of evaluating *Initializer*.
 - b. Set *v* to ? `GetValue(defaultValue)`.
4. Return the result of performing `BindingInitialization` of *BindingPattern* with *v* and *environment* as the arguments.

BindingRestElement : ... *BindingIdentifier*

1. Let *lhs* be ? `ResolveBinding(StringValue of BindingIdentifier, environment)`.
2. Let *A* be ! `ArrayCreate(0)`.
3. Let *n* be 0.
4. Repeat,
 - a. If *iteratorRecord*.[[Done]] is **false**, then
 - i. Let *next* be `IteratorStep(iteratorRecord)`.
 - ii. If *next* is an abrupt completion, set *iteratorRecord*.[[Done]] to **true**.
 - iii. `ReturnIfAbrupt(next)`.
 - iv. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
 - b. If *iteratorRecord*.[[Done]] is **true**, then
 - i. If *environment* is **undefined**, return ? `PutValue(lhs, A)`.

- ii. Return `InitializeReferencedBinding(lhs, A)`.
- c. Let `nextValue` be `IteratorValue(next)`.
- d. If `nextValue` is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
- e. `ReturnIfAbrupt(nextValue)`.
- f. Perform `! CreateDataPropertyOrThrow(A, ! ToString(F(n)), nextValue)`.
- g. Set `n` to `n + 1`.

BindingRestElement : ... *BindingPattern*

1. Let `A` be `! ArrayCreate(0)`.
2. Let `n` be 0.
3. Repeat,
 - a. If `iteratorRecord.[[Done]]` is **false**, then
 - i. Let `next` be `IteratorStep(iteratorRecord)`.
 - ii. If `next` is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
 - iii. `ReturnIfAbrupt(next)`.
 - iv. If `next` is **false**, set `iteratorRecord.[[Done]]` to **true**.
 - b. If `iteratorRecord.[[Done]]` is **true**, then
 - i. Return the result of performing `BindingInitialization` of *BindingPattern* with `A` and *environment* as the arguments.
 - c. Let `nextValue` be `IteratorValue(next)`.
 - d. If `nextValue` is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
 - e. `ReturnIfAbrupt(nextValue)`.
 - f. Perform `! CreateDataPropertyOrThrow(A, ! ToString(F(n)), nextValue)`.
 - g. Set `n` to `n + 1`.

FormalParameters : [empty]

1. Return `NormalCompletion(empty)`.

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. Perform `? IteratorBindingInitialization` for *FormalParameterList* using *iteratorRecord* and *environment* as the arguments.
2. Return the result of performing `IteratorBindingInitialization` for *FunctionRestParameter* using *iteratorRecord* and *environment* as the arguments.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. Perform `? IteratorBindingInitialization` for *FormalParameterList* using *iteratorRecord* and *environment* as the arguments.
2. Return the result of performing `IteratorBindingInitialization` for *FormalParameter* using *iteratorRecord* and *environment* as the arguments.

ArrowParameters : *BindingIdentifier*

1. **Assert**: `iteratorRecord.[[Done]]` is **false**.
2. Let `next` be `IteratorStep(iteratorRecord)`.
3. If `next` is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
4. `ReturnIfAbrupt(next)`.
5. If `next` is **false**, set `iteratorRecord.[[Done]]` to **true**.

6. Else,
 - a. Let v be `IteratorValue(next)`.
 - b. If v is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
 - c. `ReturnIfAbrupt(v)`.
7. If `iteratorRecord.[[Done]]` is **true**, let v be **undefined**.
8. Return the result of performing `BindingInitialization` for `BindingIdentifier` using v and `environment` as the arguments.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let `formals` be `CoveredFormalsList` of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return `IteratorBindingInitialization` of `formals` with arguments `iteratorRecord` and `environment`.

AsyncArrowBindingIdentifier : *BindingIdentifier*

1. **Assert**: `iteratorRecord.[[Done]]` is **false**.
2. Let `next` be `IteratorStep(iteratorRecord)`.
3. If `next` is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
4. `ReturnIfAbrupt(next)`.
5. If `next` is **false**, set `iteratorRecord.[[Done]]` to **true**.
6. Else,
 - a. Let v be `IteratorValue(next)`.
 - b. If v is an abrupt completion, set `iteratorRecord.[[Done]]` to **true**.
 - c. `ReturnIfAbrupt(v)`.
7. If `iteratorRecord.[[Done]]` is **true**, let v be **undefined**.
8. Return the result of performing `BindingInitialization` for `BindingIdentifier` using v and `environment` as the arguments.

8.5.4 Static Semantics: AssignmentTargetType

IdentifierReference : *Identifier*

1. If this *IdentifierReference* is contained in `strict mode code` and `StringValue` of *Identifier* is `"eval"` or `"arguments"`, return `invalid`.
2. Return `simple`.

IdentifierReference :

yield

await

CallExpression :

CallExpression [*Expression*]

CallExpression . *IdentifierName*

MemberExpression :

MemberExpression [*Expression*]

MemberExpression . *IdentifierName*

SuperProperty

1. Return `simple`.

PrimaryExpression :

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

PrimaryExpression :

this

Literal

ArrayLiteral

ObjectLiteral

FunctionExpression

ClassExpression

GeneratorExpression

AsyncFunctionExpression

AsyncGeneratorExpression

RegularExpressionLiteral

TemplateLiteral

CallExpression :

CoverCallExpressionAndAsyncArrowHead

SuperCall

ImportCall

CallExpression Arguments

CallExpression TemplateLiteral

NewExpression :

new *NewExpression*

MemberExpression :

MemberExpression TemplateLiteral

new *MemberExpression Arguments*

NewTarget :

new . target

ImportMeta :

import . meta

LeftHandSideExpression :

OptionalExpression

UpdateExpression :

LeftHandSideExpression **++**

LeftHandSideExpression **--**

++ *UnaryExpression*

-- *UnaryExpression*

UnaryExpression :

delete *UnaryExpression*

void *UnaryExpression*

typeof *UnaryExpression*

+ *UnaryExpression*

- *UnaryExpression*

~ UnaryExpression
 ! UnaryExpression
 AwaitExpression
 ExponentiationExpression :
 UpdateExpression ** ExponentiationExpression
 MultiplicativeExpression :
 MultiplicativeExpression MultiplicativeOperator ExponentiationExpression
 AdditiveExpression :
 AdditiveExpression + MultiplicativeExpression
 AdditiveExpression - MultiplicativeExpression
 ShiftExpression :
 ShiftExpression << AdditiveExpression
 ShiftExpression >> AdditiveExpression
 ShiftExpression >>> AdditiveExpression
 RelationalExpression :
 RelationalExpression < ShiftExpression
 RelationalExpression > ShiftExpression
 RelationalExpression <= ShiftExpression
 RelationalExpression >= ShiftExpression
 RelationalExpression instanceof ShiftExpression
 RelationalExpression in ShiftExpression
 EqualityExpression :
 EqualityExpression == RelationalExpression
 EqualityExpression != RelationalExpression
 EqualityExpression === RelationalExpression
 EqualityExpression !== RelationalExpression
 BitwiseANDExpression :
 BitwiseANDExpression & EqualityExpression
 BitwiseXORExpression :
 BitwiseXORExpression ^ BitwiseANDExpression
 BitwiseORExpression :
 BitwiseORExpression | BitwiseXORExpression
 LogicalANDExpression :
 LogicalANDExpression && BitwiseORExpression
 LogicalORExpression :
 LogicalORExpression || LogicalANDExpression
 CoalesceExpression :
 CoalesceExpressionHead ?? BitwiseORExpression
 ConditionalExpression :
 ShortCircuitExpression ? AssignmentExpression : AssignmentExpression
 AssignmentExpression :
 YieldExpression
 ArrowFunction
 AsyncArrowFunction
 LeftHandSideExpression = AssignmentExpression

LeftHandSideExpression AssignmentOperator AssignmentExpression

LeftHandSideExpression &&= AssignmentExpression

LeftHandSideExpression ||= AssignmentExpression

LeftHandSideExpression ??= AssignmentExpression

Expression :

Expression , AssignmentExpression

1. Return `invalid`.

8.5.5 Static Semantics: `PropName`

PropertyDefinition : IdentifierReference

1. Return `StringValue` of *IdentifierReference*.

PropertyDefinition : ... AssignmentExpression

1. Return `empty`.

PropertyDefinition : PropertyName : AssignmentExpression

1. Return `PropName` of *PropertyName*.

LiteralPropertyName : IdentifierName

1. Return `StringValue` of *IdentifierName*.

LiteralPropertyName : StringLiteral

1. Return the `SV` of *StringLiteral*.

LiteralPropertyName : NumericLiteral

1. Let *nbr* be the `NumericValue` of *NumericLiteral*.
2. Return `! ToString(nbr)`.

ComputedPropertyName : [AssignmentExpression]

1. Return `empty`.

MethodDefinition :

PropertyName (UniqueFormalParameters) { FunctionBody }

get *PropertyName () { FunctionBody }*

set *PropertyName (PropertySetParameterList) { FunctionBody }*

1. Return `PropName` of *PropertyName*.

*GeneratorMethod : * PropertyName (UniqueFormalParameters) { GeneratorBody }*

1. Return `PropName` of *PropertyName*.

*AsyncGeneratorMethod : **async** * PropertyName (UniqueFormalParameters) { AsyncGeneratorBody }*

1. Return `PropName` of *PropertyName*.

ClassElement : ;

1. Return empty.

AsyncMethod : **async** *PropertyName* (*UniqueFormalParameters*) { *AsyncFunctionBody* }

1. Return **PropName** of *PropertyName*.

9 Executable Code and Execution Contexts

9.1 Environment Records

Environment Record 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. Usually an Environment Record is associated with some specific syntactic structure of ECMAScript code such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch* clause of a *TryStatement*. Each time such code is evaluated, a new Environment Record is created to record the identifier bindings that are created by that code.

Every Environment Record has an `[[OuterEnv]]` field, which is either **null** or a reference to an outer Environment Record. This is used to model the logical nesting of Environment Record values. The outer reference of an (inner) Environment Record is a reference to the Environment Record that logically surrounds the inner Environment Record. An outer Environment Record may, of course, have its own outer Environment Record. An Environment Record may serve as the outer environment for multiple inner Environment Records. For example, if a *FunctionDeclaration* contains two nested *FunctionDeclarations* then the Environment Records of each of the nested functions will have as their outer Environment Record the Environment Record of the current evaluation of the surrounding function.

Environment Records 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.

9.1.1 The Environment Record Type Hierarchy

Environment Records 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 and module Environment Records are subclasses of **declarative Environment Record**.

- **Environment Record** (abstract)
 - A *declarative Environment Record* is 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.
 - A *function Environment Record* corresponds to the invocation of an ECMAScript **function object**, and contains bindings for the top-level declarations within that function. It may establish a new **this** binding. It also captures the state necessary to support **super** method invocations.
 - A *module Environment Record* contains the bindings for the top-level declarations of a *Module*. It

also contains the bindings that are explicitly imported by the *Module*. Its `[[OuterEnv]]` is a [global Environment Record](#).

- An *object Environment Record* is used to define the effect of ECMAScript elements such as *WithStatement* that associate identifier bindings with the properties of some object.
- A *global Environment Record* is used for *Script* global declarations. It does not have an outer environment; its `[[OuterEnv]]` is **null**. It may be prepopulated with identifier bindings and it includes an associated [global object](#) whose properties provide some of the global environment's identifier bindings. As ECMAScript code is executed, additional properties may be added to the [global object](#) and the initial properties may be modified.

The [Environment Record](#) abstract class includes the abstract specification methods defined in [Table 17](#). These abstract methods have distinct concrete algorithms for each of the concrete subclasses.

Table 17: Abstract Methods of Environment Records

Method	Purpose
HasBinding(N)	Determine if an Environment Record has a binding for the String value <i>N</i> . Return true if it does and false if it does not.
CreateMutableBinding(N, D)	Create a new but uninitialized mutable binding in an Environment Record . The String value <i>N</i> is the text of the bound name. If the Boolean argument <i>D</i> is true the binding may be subsequently deleted.
CreateImmutableBinding(N, S)	Create a new but uninitialized immutable binding in an Environment Record . The String value <i>N</i> is the text of the bound name. If <i>S</i> is true then attempts to set it after it has been initialized will always throw an exception, regardless of the strict mode setting of operations that reference that binding.
InitializeBinding(N, V)	Set the value of an already existing but uninitialized binding in an Environment Record . The String value <i>N</i> is the text of the bound name. <i>V</i> is the value for the binding and is a value of any ECMAScript language type .
SetMutableBinding(N, V, S)	Set the value of an already existing mutable binding in an Environment Record . The String value <i>N</i> is the text of the bound name. <i>V</i> is the value for the binding and may be a value of any ECMAScript language type . <i>S</i> is a Boolean flag. If <i>S</i> is true and the binding cannot be set throw a TypeError exception.
GetBindingValue(N, S)	Returns the value of an already existing binding from an Environment Record . The String value <i>N</i> is the text of the bound name. <i>S</i> is used to identify references originating in strict mode code or that otherwise require strict mode reference semantics. If <i>S</i> is true and the binding does not exist throw a ReferenceError exception. If the binding exists but is uninitialized a ReferenceError is thrown, regardless of the value of <i>S</i> .
DeleteBinding(N)	Delete a binding from an Environment Record . The String value <i>N</i> is the text of the bound name. If a binding for <i>N</i> exists, remove the binding and return true . If the binding exists but cannot be removed return false . If the binding does not exist return true .
HasThisBinding()	Determine if an Environment Record establishes a this binding. Return true if it does and false if it does not.
HasSuperBinding()	Determine if an Environment Record establishes a super method binding. Return true if it does and false if it does not.
WithBaseObject()	If this Environment Record is associated with a with statement, return the with object. Otherwise, return undefined .

9.1.1.1 Declarative Environment Records

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

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

9.1.1.1.1 HasBinding (*N*)

The HasBinding concrete method of a [declarative Environment Record](#) *envRec* takes argument *N* (a String). It determines if the argument identifier is one of the identifiers bound by the record. It performs the following steps when called:

1. If *envRec* has a binding for the name that is the value of *N*, return **true**.
2. Return **false**.

9.1.1.1.2 CreateMutableBinding (*N*, *D*)

The CreateMutableBinding concrete method of a [declarative Environment Record](#) *envRec* takes arguments *N* (a String) and *D* (a Boolean). It 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 *D* has the value **true**, the new binding is marked as being subject to deletion. It performs the following steps when called:

1. **Assert**: *envRec* does not already have a binding for *N*.
2. Create a mutable binding in *envRec* for *N* and record that it is uninitialized. If *D* is **true**, record that the newly created binding may be deleted by a subsequent DeleteBinding call.
3. Return [NormalCompletion](#)(empty).

9.1.1.1.3 CreateImmutableBinding (*N*, *S*)

The CreateImmutableBinding concrete method of a [declarative Environment Record](#) *envRec* takes arguments *N* (a String) and *S* (a Boolean). It creates a new immutable binding for the name *N* that is uninitialized. A binding must not already exist in this [Environment Record](#) for *N*. If *S* has the value **true**, the new binding is marked as a strict binding. It performs the following steps when called:

1. **Assert**: *envRec* does not already have a binding for *N*.
2. Create an immutable binding in *envRec* for *N* and record that it is uninitialized. If *S* is **true**, record that the newly created binding is a strict binding.
3. Return [NormalCompletion](#)(empty).

9.1.1.1.4 InitializeBinding (*N*, *V*)

The InitializeBinding concrete method of a [declarative Environment Record](#) *envRec* takes arguments *N* (a String) and *V* (an [ECMAScript language value](#)). It 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. It performs the following steps when called:

1. **Assert**: *envRec* must have an uninitialized binding for *N*.
2. Set the bound value for *N* in *envRec* to *V*.
3. Record that the binding for *N* in *envRec* has been initialized.
4. Return [NormalCompletion](#)(empty).

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

The SetMutableBinding concrete method of a [declarative Environment Record](#) *envRec* takes arguments *N* (a String), *V*

(an [ECMAScript language value](#)), and S (a Boolean). It 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 normally already exists, but in rare cases it may not. If the binding is an immutable binding, a **TypeError** is thrown if S is **true**. It performs the following steps when called:

1. If *envRec* does not have a binding for N , then
 - a. If S is **true**, throw a **ReferenceError** exception.
 - b. Perform *envRec*.CreateMutableBinding(N , **true**).
 - c. Perform *envRec*.InitializeBinding(N , V).
 - d. Return [NormalCompletion](#)(empty).
2. If the binding for N in *envRec* is a strict binding, set S to **true**.
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,
 - a. **Assert**: This is an attempt to change the value of an immutable binding.
 - b. If S is **true**, throw a **TypeError** exception.
6. Return [NormalCompletion](#)(empty).

NOTE An example of ECMAScript code that results in a missing binding at step 1 is:

```
function f() { eval("var x; x = (delete x, 0);"); }
```

9.1.1.1.6 GetBindingValue (N , S)

The GetBindingValue concrete method of a [declarative Environment Record](#) *envRec* takes arguments N (a String) and S (a Boolean). It 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 . It performs the following steps when called:

1. **Assert**: *envRec* has a binding for N .
2. If the binding for N in *envRec* is an uninitialized binding, throw a **ReferenceError** exception.
3. Return the value currently bound to N in *envRec*.

9.1.1.1.7 DeleteBinding (N)

The DeleteBinding concrete method of a [declarative Environment Record](#) *envRec* takes argument N (a String). It can only delete bindings that have been explicitly designated as being subject to deletion. It performs the following steps when called:

1. **Assert**: *envRec* has a binding for the name that is the value of N .
2. If the binding for N in *envRec* cannot be deleted, return **false**.
3. Remove the binding for N from *envRec*.
4. Return **true**.

9.1.1.1.8 HasThisBinding ()

The HasThisBinding concrete method of a [declarative Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **false**.

NOTE A regular [declarative Environment Record](#) (i.e., one that is neither a [function Environment Record](#) nor a [module Environment Record](#)) does not provide a **this** binding.

9.1.1.1.9 HasSuperBinding ()

The HasSuperBinding concrete method of a [declarative Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **false**.

NOTE A regular [declarative Environment Record](#) (i.e., one that is neither a [function Environment Record](#) nor a [module Environment Record](#)) does not provide a **super** binding.

9.1.1.1.10 WithBaseObject ()

The WithBaseObject concrete method of a [declarative Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **undefined**.

9.1.1.2 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 created for **with** statements (14.11) 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.

9.1.1.2.1 HasBinding (*N*)

The HasBinding concrete method of an [object Environment Record](#) *envRec* takes argument *N* (a String). It determines if its associated binding object has a property whose name is the value of the argument *N*. It performs the following steps when called:

1. Let *bindings* be the binding object for *envRec*.
2. Let *foundBinding* be ? `HasProperty(bindings, N)`.
3. If *foundBinding* is **false**, return **false**.

4. If the *withEnvironment* flag of *envRec* is **false**, return **true**.
5. Let *unscopables* be ? *Get(bindings, @@unscopables)*.
6. If *Type(unscopables)* is **Object**, then
 - a. Let *blocked* be ! *ToBoolean(? Get(unscopables, N))*.
 - b. If *blocked* is **true**, return **false**.
7. Return **true**.

9.1.1.2.2 CreateMutableBinding (*N*, *D*)

The *CreateMutableBinding* concrete method of an **object Environment Record** *envRec* takes arguments *N* (a String) and *D* (a Boolean). It 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 *D* has the value **true**, the new property's **[[Configurable]]** attribute is set to **true**; otherwise it is set to **false**. It performs the following steps when called:

1. Let *bindings* be the binding object for *envRec*.
2. Return ? *DefinePropertyOrThrow(bindings, N, PropertyDescriptor { [[Value]]: undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: *D* })*.

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.

9.1.1.2.3 CreateImmutableBinding (*N*, *S*)

The *CreateImmutableBinding* concrete method of an **object Environment Record** is never used within this specification.

9.1.1.2.4 InitializeBinding (*N*, *V*)

The *InitializeBinding* concrete method of an **object Environment Record** *envRec* takes arguments *N* (a String) and *V* (an **ECMAScript language value**). It 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*. It performs the following steps when called:

1. Return ? *envRec.SetMutableBinding(N, V, false)*.

NOTE In this specification, all uses of *CreateMutableBinding* for **object Environment Records** are immediately followed by a call to *InitializeBinding* for the same name. Hence, this specification does not explicitly track the initialization state of bindings in **object Environment Records**.

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

The *SetMutableBinding* concrete method of an **object Environment Record** *envRec* takes arguments *N* (a String), *V* (an **ECMAScript language value**), and *S* (a Boolean). It 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 *S*. It performs the following steps when called:

1. Let *bindings* be the binding object for *envRec*.
2. Let *stillExists* be ? *HasProperty(bindings, N)*.

3. If *stillExists* is **false** and *S* is **true**, throw a **ReferenceError** exception.
4. Return ? *Set(bindings, N, V, S)*.

9.1.1.2.6 GetBindingValue (*N*, *S*)

The *GetBindingValue* concrete method of an **object Environment Record** *envRec* takes arguments *N* (a String) and *S* (a Boolean). It 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 *S*. It performs the following steps when called:

1. Let *bindings* be the binding object for *envRec*.
2. Let *value* be ? *HasProperty(bindings, N)*.
3. If *value* is **false**, then
 - a. If *S* is **false**, return the value **undefined**; otherwise throw a **ReferenceError** exception.
4. Return ? *Get(bindings, N)*.

9.1.1.2.7 DeleteBinding (*N*)

The *DeleteBinding* concrete method of an **object Environment Record** *envRec* takes argument *N* (a String). It can only delete bindings that correspond to properties of the environment object whose *[[Configurable]]* attribute have the value **true**. It performs the following steps when called:

1. Let *bindings* be the binding object for *envRec*.
2. Return ? *bindings*.*[[Delete]]*(*N*).

9.1.1.2.8 HasThisBinding ()

The *HasThisBinding* concrete method of an **object Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. Return **false**.

NOTE Object Environment Records do not provide a **this** binding.

9.1.1.2.9 HasSuperBinding ()

The *HasSuperBinding* concrete method of an **object Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. Return **false**.

NOTE Object Environment Records do not provide a **super** binding.

9.1.1.2.10 WithBaseObject ()

The *WithBaseObject* concrete method of an **object Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. If the *withEnvironment* flag of *envRec* is **true**, return the binding object for *envRec*.
2. Otherwise, return **undefined**.

9.1.1.3 Function Environment Records

A *function Environment Record* is a [declarative Environment Record](#) that is used to represent the top-level scope of a function and, if the function is not an *ArrowFunction*, provides a **this** binding. If a function is not an *ArrowFunction* function and 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 have the additional state fields listed in [Table 18](#).

Table 18: Additional Fields of Function Environment Records

Field Name	Value	Meaning
[[ThisValue]]	Any	This is the this value used for this invocation of the function.
[[ThisBindingStatus]]	lexical initialized uninitialized	If the value is lexical , this is an <i>ArrowFunction</i> and does not have a local this value.
[[FunctionObject]]	Object	The function object whose invocation caused this Environment Record to be created.
[[NewTarget]]	Object undefined	If this Environment Record was created by the [[Construct]] internal method, [[NewTarget]] is the value of the [[Construct]] <i>newTarget</i> parameter. Otherwise, its value is undefined .

Function Environment Records support all of the [declarative Environment Record](#) methods listed in [Table 17](#) and share the same specifications for all of those methods except for `HasThisBinding` and `HasSuperBinding`. In addition, function Environment Records support the methods listed in [Table 19](#):

Table 19: Additional Methods of Function Environment Records

Method	Purpose
<code>BindThisValue(V)</code>	Set the [[ThisValue]] and record that it has been initialized.
<code>GetThisBinding()</code>	Return the value of this Environment Record 's this binding. Throws a ReferenceError if the this binding has not been initialized.
<code>GetSuperBase()</code>	Return the object that is the base for super property accesses bound in this Environment Record . The value undefined indicates that super property accesses will produce runtime errors.

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

9.1.1.3.1 BindThisValue (V)

The `BindThisValue` concrete method of a [function Environment Record](#) *envRec* takes argument *V* (an [ECMAScript language value](#)). It performs the following steps when called:

1. **Assert:** *envRec*.[[ThisBindingStatus]] is not **lexical**.
2. If *envRec*.[[ThisBindingStatus]] is initialized, throw a **ReferenceError** exception.
3. Set *envRec*.[[ThisValue]] to *V*.
4. Set *envRec*.[[ThisBindingStatus]] to initialized.
5. Return *V*.

9.1.1.3.2 HasThisBinding ()

The HasThisBinding concrete method of a **function Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. If *envRec*.[[ThisBindingStatus]] is **lexical**, return **false**; otherwise, return **true**.

9.1.1.3.3 HasSuperBinding ()

The HasSuperBinding concrete method of a **function Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. If *envRec*.[[ThisBindingStatus]] is **lexical**, return **false**.
2. If *envRec*.[[FunctionObject]].[[HomeObject]] has the value **undefined**, return **false**; otherwise, return **true**.

9.1.1.3.4 GetThisBinding ()

The GetThisBinding concrete method of a **function Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. **Assert:** *envRec*.[[ThisBindingStatus]] is not **lexical**.
2. If *envRec*.[[ThisBindingStatus]] is uninitialized, throw a **ReferenceError** exception.
3. Return *envRec*.[[ThisValue]].

9.1.1.3.5 GetSuperBase ()

The GetSuperBase concrete method of a **function Environment Record** *envRec* takes no arguments. It performs the following steps when called:

1. Let *home* be *envRec*.[[FunctionObject]].[[HomeObject]].
2. If *home* has the value **undefined**, return **undefined**.
3. **Assert:** *Type(home)* is **Object**.
4. Return ? *home*.[[GetPrototypeOf]]().

9.1.1.4 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**. A global Environment Record provides the bindings for built-in globals (clause 19), properties of the **global object**, and for all top-level declarations (8.1.9, 8.1.11) that occur within a *Script*.

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 Record**. This **global object** is the value returned by the global Environment Record's GetThisBinding concrete method. The **object Environment Record** component of a global Environment

Record contains the bindings for all built-in globals (clause 19) and all bindings introduced by a *FunctionDeclaration*, *GeneratorDeclaration*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, 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*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, or *VariableDeclaration* declarations and bindings 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 `CreateGlobalVarBinding` and `CreateGlobalFunctionBinding` concrete methods.

Global Environment Records have the additional fields listed in Table 20 and the additional methods listed in Table 21.

Table 20: Additional Fields of Global Environment Records

Field Name	Value	Meaning
[[ObjectRecord]]	Object Environment Record	Binding object is the global object . It contains global built-in bindings as well as <i>FunctionDeclaration</i> , <i>GeneratorDeclaration</i> , <i>AsyncFunctionDeclaration</i> , <i>AsyncGeneratorDeclaration</i> , and <i>VariableDeclaration</i> bindings in global code for the associated realm .
[[GlobalThisValue]]	Object	The value returned by this in global scope. Hosts may provide any ECMAScript Object value.
[[DeclarativeRecord]]	Declarative Environment Record	Contains bindings for all declarations in global code for the associated realm code except for <i>FunctionDeclaration</i> , <i>GeneratorDeclaration</i> , <i>AsyncFunctionDeclaration</i> , <i>AsyncGeneratorDeclaration</i> , and <i>VariableDeclaration</i> bindings.
[[VarNames]]	List of String	The string names bound by <i>FunctionDeclaration</i> , <i>GeneratorDeclaration</i> , <i>AsyncFunctionDeclaration</i> , <i>AsyncGeneratorDeclaration</i> , and <i>VariableDeclaration</i> declarations in global code for the associated realm .

Table 21: Additional Methods of Global Environment Records

Method	Purpose
GetThisBinding()	Return the value of this Environment Record 's this binding.
HasVarDeclaration (N)	Determines if the argument identifier has a binding in this Environment Record that was created using a <i>VariableDeclaration</i> , <i>FunctionDeclaration</i> , <i>GeneratorDeclaration</i> , <i>AsyncFunctionDeclaration</i> , or <i>AsyncGeneratorDeclaration</i> .
HasLexicalDeclaration (N)	Determines if the argument identifier has a binding in this Environment Record that was created using a lexical declaration such as a <i>LexicalDeclaration</i> or a <i>ClassDeclaration</i> .
HasRestrictedGlobalProperty (N)	Determines if the argument is the name of a global object property that may not be shadowed by a global lexical binding.
CanDeclareGlobalVar (N)	Determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument <i>N</i> .
CanDeclareGlobalFunction (N)	Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument <i>N</i> .
CreateGlobalVarBinding(N, D)	Used to create and initialize to undefined a global var binding 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 <i>N</i> is the bound name. If <i>D</i> is true the binding may be deleted. Logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows var declarations to receive special treatment.
CreateGlobalFunctionBinding(N, V, D)	Create and initialize a global function binding 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 <i>N</i> is the bound name. <i>V</i> is the initialization value. If the Boolean argument <i>D</i> is true the binding may be deleted. 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.

9.1.1.4.1 HasBinding (*N*)

The HasBinding concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It determines if the argument identifier is one of the identifiers bound by the record. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, return **true**.

3. Let *ObjRec* be *envRec*.[[ObjectRecord]].
4. Return ? *ObjRec*.HasBinding(*N*).

9.1.1.4.2 CreateMutableBinding (*N*, *D*)

The CreateMutableBinding concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String) and *D* (a Boolean). It 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 *D* has the value **true**, the new binding is marked as being subject to deletion. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, throw a **TypeError** exception.
3. Return *DclRec*.CreateMutableBinding(*N*, *D*).

9.1.1.4.3 CreateImmutableBinding (*N*, *S*)

The CreateImmutableBinding concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String) and *S* (a Boolean). It creates a new immutable binding for the name *N* that is uninitialized. A binding must not already exist in this [Environment Record](#) for *N*. If *S* has the value **true**, the new binding is marked as a strict binding. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, throw a **TypeError** exception.
3. Return *DclRec*.CreateImmutableBinding(*N*, *S*).

9.1.1.4.4 InitializeBinding (*N*, *V*)

The InitializeBinding concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String) and *V* (an [ECMAScript language value](#)). It 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. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, then
 - a. Return *DclRec*.InitializeBinding(*N*, *V*).
3. **Assert**: If the binding exists, it must be in the [object Environment Record](#).
4. Let *ObjRec* be *envRec*.[[ObjectRecord]].
5. Return ? *ObjRec*.InitializeBinding(*N*, *V*).

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

The SetMutableBinding concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String), *V* (an [ECMAScript language value](#)), and *S* (a Boolean). It 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 *S*. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, then
 - a. Return *DclRec*.SetMutableBinding(*N*, *V*, *S*).

3. Let *ObjRec* be *envRec*.[[ObjectRecord]].
4. Return ? *ObjRec*.SetMutableBinding(*N*, *V*, *S*).

9.1.1.4.6 GetBindingValue (*N*, *S*)

The GetBindingValue concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String) and *S* (a Boolean). It 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 *S*. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, then
 - a. Return *DclRec*.GetBindingValue(*N*, *S*).
3. Let *ObjRec* be *envRec*.[[ObjectRecord]].
4. Return ? *ObjRec*.GetBindingValue(*N*, *S*).

9.1.1.4.7 DeleteBinding (*N*)

The DeleteBinding concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It can only delete bindings that have been explicitly designated as being subject to deletion. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. If *DclRec*.HasBinding(*N*) is **true**, then
 - a. Return *DclRec*.DeleteBinding(*N*).
3. Let *ObjRec* be *envRec*.[[ObjectRecord]].
4. Let *globalObject* be the binding object for *ObjRec*.
5. Let *existingProp* be ? [HasOwnProperty](#)(*globalObject*, *N*).
6. If *existingProp* is **true**, then
 - a. Let *status* be ? *ObjRec*.DeleteBinding(*N*).
 - b. If *status* is **true**, then
 - i. Let *varNames* be *envRec*.[[VarNames]].
 - ii. If *N* is an element of *varNames*, remove that element from the *varNames*.
 - c. Return *status*.
7. Return **true**.

9.1.1.4.8 HasThisBinding ()

The HasThisBinding concrete method of a [global Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **true**.

NOTE Global Environment Records always provide a **this** binding.

9.1.1.4.9 HasSuperBinding ()

The HasSuperBinding concrete method of a [global Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **false**.

NOTE Global Environment Records do not provide a **super** binding.

9.1.1.4.10 WithBaseObject ()

The WithBaseObject concrete method of a [global Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **undefined**.

9.1.1.4.11 GetThisBinding ()

The GetThisBinding concrete method of a [global Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return *envRec*.[[GlobalThisValue]].

9.1.1.4.12 HasVarDeclaration (*N*)

The HasVarDeclaration concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It determines if the argument identifier has a binding in this record that was created using a *VariableStatement* or a *FunctionDeclaration*. It performs the following steps when called:

1. Let *varDeclaredNames* be *envRec*.[[VarNames]].
2. If *varDeclaredNames* contains *N*, return **true**.
3. Return **false**.

9.1.1.4.13 HasLexicalDeclaration (*N*)

The HasLexicalDeclaration concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It 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*. It performs the following steps when called:

1. Let *DclRec* be *envRec*.[[DeclarativeRecord]].
2. Return *DclRec*.HasBinding(*N*).

9.1.1.4.14 HasRestrictedGlobalProperty (*N*)

The HasRestrictedGlobalProperty concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It determines if the argument identifier is the name of a property of the [global object](#) that must not be shadowed by a global lexical binding. It performs the following steps when called:

1. Let *ObjRec* be *envRec*.[[ObjectRecord]].
2. Let *globalObject* be the binding object for *ObjRec*.
3. Let *existingProp* be ? *globalObject*.[[GetOwnProperty]](*N*).
4. If *existingProp* is **undefined**, return **false**.
5. If *existingProp*.[[Configurable]] is **true**, return **false**.
6. Return **true**.

NOTE

Properties may exist upon a [global object](#) that were directly created rather than being declared using a var or function declaration. A global lexical binding may not be created that has the same name as a non-configurable property of the [global object](#). The global property **"undefined"** is an example of such a property.

9.1.1.4.15 CanDeclareGlobalVar (*N*)

The CanDeclareGlobalVar concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It 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. It performs the following steps when called:

1. Let *ObjRec* be *envRec*.[[ObjectRecord]].
2. Let *globalObject* be the binding object for *ObjRec*.
3. Let *hasProperty* be ? [HasOwnProperty](#)(*globalObject*, *N*).
4. If *hasProperty* is **true**, return **true**.
5. Return ? [IsExtensible](#)(*globalObject*).

9.1.1.4.16 CanDeclareGlobalFunction (*N*)

The CanDeclareGlobalFunction concrete method of a [global Environment Record](#) *envRec* takes argument *N* (a String). It determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument *N*. It performs the following steps when called:

1. Let *ObjRec* be *envRec*.[[ObjectRecord]].
2. Let *globalObject* be the binding object for *ObjRec*.
3. Let *existingProp* be ? *globalObject*.[[GetOwnProperty]](*N*).
4. If *existingProp* is **undefined**, return ? [IsExtensible](#)(*globalObject*).
5. If *existingProp*.[[Configurable]] is **true**, return **true**.
6. If [IsDataDescriptor](#)(*existingProp*) is **true** and *existingProp* has attribute values { [[Writable]]: **true**, [[Enumerable]]: **true** }, return **true**.
7. Return **false**.

9.1.1.4.17 CreateGlobalVarBinding (*N*, *D*)

The CreateGlobalVarBinding concrete method of a [global Environment Record](#) *envRec* takes arguments *N* (a String) and *D* (a Boolean). It creates and initializes 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 and assumed to be initialized. It performs the following steps when called:

1. Let *ObjRec* be *envRec*.[[ObjectRecord]].
2. Let *globalObject* be the binding object for *ObjRec*.
3. Let *hasProperty* be ? [HasOwnProperty](#)(*globalObject*, *N*).
4. Let *extensible* be ? [IsExtensible](#)(*globalObject*).
5. If *hasProperty* is **false** and *extensible* is **true**, then
 - a. Perform ? *ObjRec*.CreateMutableBinding(*N*, *D*).
 - b. Perform ? *ObjRec*.InitializeBinding(*N*, **undefined**).
6. Let *varDeclaredNames* be *envRec*.[[VarNames]].
7. If *varDeclaredNames* does not contain *N*, then

- a. Append *N* to *varDeclaredNames*.
8. Return `NormalCompletion(empty)`.

9.1.1.4.18 CreateGlobalFunctionBinding (*N*, *V*, *D*)

The `CreateGlobalFunctionBinding` concrete method of a `global Environment Record` *envRec* takes arguments *N* (a String), *V* (an `ECMAScript language value`), and *D* (a Boolean). It creates and initializes 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. It performs the following steps when called:

1. Let *ObjRec* be *envRec*.`[[ObjectRecord]]`.
2. Let *globalObject* be the binding object for *ObjRec*.
3. Let *existingProp* be `? globalObject. [[GetOwnProperty]](N)`.
4. If *existingProp* is **undefined** or *existingProp*.`[[Configurable]]` is **true**, then
 - a. Let *desc* be the `PropertyDescriptor` { `[[Value]]: V`, `[[Writable]]: true`, `[[Enumerable]]: true`, `[[Configurable]]: D` }.
5. Else,
 - a. Let *desc* be the `PropertyDescriptor` { `[[Value]]: V` }.
6. Perform `? DefinePropertyOrThrow(globalObject, N, desc)`.
7. Perform `? Set(globalObject, N, V, false)`.
8. Let *varDeclaredNames* be *envRec*.`[[VarNames]]`.
9. If *varDeclaredNames* does not contain *N*, then
 - a. Append *N* to *varDeclaredNames*.
10. 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. Step 7 is equivalent to what calling the `InitializeBinding` concrete method would do and if *globalObject* is a Proxy will produce the same sequence of Proxy trap calls.

9.1.1.5 Module Environment Records

A *module Environment Record* is a `declarative Environment Record` that is used to represent the outer scope of an `ECMAScript Module`. In addition to normal mutable and immutable bindings, module Environment Records also provide immutable import bindings which are bindings that provide indirect access to a target binding that exists in another `Environment Record`.

Module Environment Records support all of the `declarative Environment Record` methods listed in [Table 17](#) and share the same specifications for all of those methods except for `GetBindingValue`, `DeleteBinding`, `HasThisBinding` and `GetThisBinding`. In addition, module Environment Records support the methods listed in [Table 22](#):

Table 22: Additional Methods of Module Environment Records

Method	Purpose
CreateImportBinding(<i>N</i> , <i>M</i> , <i>N2</i>)	Create an immutable indirect binding in a module Environment Record . The String value <i>N</i> is the text of the bound name. <i>M</i> is a Module Record , and <i>N2</i> is a binding that exists in <i>M</i> 's module Environment Record .
GetThisBinding()	Return the value of this Environment Record 's this binding.

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

9.1.1.5.1 GetBindingValue (*N*, *S*)

The GetBindingValue concrete method of a [module Environment Record](#) *envRec* takes arguments *N* (a String) and *S* (a Boolean). It returns the value of its bound identifier whose name is the value of the argument *N*. However, if the binding is an indirect binding the value of the target binding is returned. If the binding exists but is uninitialized a **ReferenceError** is thrown. It performs the following steps when called:

1. **Assert**: *S* is **true**.
2. **Assert**: *envRec* has a binding for *N*.
3. If the binding for *N* is an indirect binding, then
 - a. Let *M* and *N2* be the indirection values provided when this binding for *N* was created.
 - b. Let *targetEnv* be *M*.[[Environment]].
 - c. If *targetEnv* is **undefined**, throw a **ReferenceError** exception.
 - d. Return ? *targetEnv*.GetBindingValue(*N2*, **true**).
4. If the binding for *N* in *envRec* is an uninitialized binding, throw a **ReferenceError** exception.
5. Return the value currently bound to *N* in *envRec*.

NOTE *S* will always be **true** because a *Module* is always [strict mode code](#).

9.1.1.5.2 DeleteBinding (*N*)

The DeleteBinding concrete method of a [module Environment Record](#) is never used within this specification.

NOTE Module Environment Records are only used within strict code and an [early error](#) rule prevents the delete operator, in strict code, from being applied to a [Reference Record](#) that would resolve to a [module Environment Record](#) binding. See 13.5.1.1.

9.1.1.5.3 HasThisBinding ()

The HasThisBinding concrete method of a [module Environment Record](#) *envRec* takes no arguments. It performs the following steps when called:

1. Return **true**.

9.1.1.5.4 GetThisBinding ()

The `GetThisBinding` concrete method of a `module Environment Record` *envRec* takes no arguments. It performs the following steps when called:

1. Return **undefined**.

9.1.1.5.5 CreateImportBinding (*N*, *M*, *N2*)

The `CreateImportBinding` concrete method of a `module Environment Record` *envRec* takes arguments *N* (a String), *M* (a `Module Record`), and *N2* (a String). It creates a new initialized immutable indirect binding for the name *N*. A binding must not already exist in this `Environment Record` for *N*. *N2* is the name of a binding that exists in *M*'s `module Environment Record`. Accesses to the value of the new binding will indirectly access the bound value of the target binding. It performs the following steps when called:

1. **Assert:** *envRec* does not already have a binding for *N*.
2. **Assert:** *M* is a `Module Record`.
3. **Assert:** When *M*.[[`Environment`]] is instantiated it will have a direct binding for *N2*.
4. Create an immutable indirect binding in *envRec* for *N* that references *M* and *N2* as its target binding and record that the binding is initialized.
5. Return `NormalCompletion(empty)`.

9.1.2 Environment Record Operations

The following `abstract operations` are used in this specification to operate upon `Environment Records`:

9.1.2.1 GetIdentifierReference (*env*, *name*, *strict*)

The abstract operation `GetIdentifierReference` takes arguments *env* (an `Environment Record` or **null**), *name* (a String), and *strict* (a Boolean). It performs the following steps when called:

1. If *env* is the value **null**, then
 - a. Return the `Reference Record` { [[`Base`]]: **unresolvable**, [[`ReferencedName`]]: *name*, [[`Strict`]]: *strict*, [[`ThisValue`]]: **empty** }.
2. Let *exists* be ? *env*.HasBinding(*name*).
3. If *exists* is **true**, then
 - a. Return the `Reference Record` { [[`Base`]]: *env*, [[`ReferencedName`]]: *name*, [[`Strict`]]: *strict*, [[`ThisValue`]]: **empty** }.
4. Else,
 - a. Let *outer* be *env*.[[`OuterEnv`]].
 - b. Return ? `GetIdentifierReference(outer, name, strict)`.

9.1.2.2 NewDeclarativeEnvironment (*E*)

The abstract operation `NewDeclarativeEnvironment` takes argument *E* (an `Environment Record`). It performs the following steps when called:

1. Let *env* be a new **declarative Environment Record** containing no bindings.
2. Set *env*.[[OuterEnv]] to *E*.
3. Return *env*.

9.1.2.3 NewObjectEnvironment (*O*, *E*)

The abstract operation NewObjectEnvironment takes arguments *O* (an Object) and *E* (an **Environment Record**). It performs the following steps when called:

1. Let *env* be a new **object Environment Record** containing *O* as the binding object.
2. Set *env*.[[OuterEnv]] to *E*.
3. Return *env*.

9.1.2.4 NewFunctionEnvironment (*F*, *newTarget*)

The abstract operation NewFunctionEnvironment takes arguments *F* and *newTarget*. It performs the following steps when called:

1. **Assert**: *F* is an ECMAScript function.
2. **Assert**: **Type**(*newTarget*) is Undefined or Object.
3. Let *env* be a new **function Environment Record** containing no bindings.
4. Set *env*.[[FunctionObject]] to *F*.
5. If *F*.[[ThisMode]] is **lexical**, set *env*.[[ThisBindingStatus]] to **lexical**.
6. Else, set *env*.[[ThisBindingStatus]] to **uninitialized**.
7. Set *env*.[[NewTarget]] to *newTarget*.
8. Set *env*.[[OuterEnv]] to *F*.[[Environment]].
9. Return *env*.

9.1.2.5 NewGlobalEnvironment (*G*, *thisValue*)

The abstract operation NewGlobalEnvironment takes arguments *G* and *thisValue*. It performs the following steps when called:

1. Let *objRec* be a new **object Environment Record** containing *G* as the binding object.
2. Let *dclRec* be a new **declarative Environment Record** containing no bindings.
3. Let *env* be a new **global Environment Record**.
4. Set *env*.[[ObjectRecord]] to *objRec*.
5. Set *env*.[[GlobalThisValue]] to *thisValue*.
6. Set *env*.[[DeclarativeRecord]] to *dclRec*.
7. Set *env*.[[VarNames]] to a new empty **List**.
8. Set *env*.[[OuterEnv]] to **null**.
9. Return *env*.

9.1.2.6 NewModuleEnvironment (*E*)

The abstract operation NewModuleEnvironment takes argument *E* (an **Environment Record**). It performs the following steps when called:

1. Let *env* be a new **module Environment Record** containing no bindings.

2. Set *env*.[[OuterEnv]] to *E*.
3. Return *env*.

9.2 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, and other associated state and resources.

A *realm* is represented in this specification as a *Realm Record* with the fields specified in Table 23:

Table 23: **Realm Record Fields**

Field Name	Value	Meaning
[[Intrinsics]]	<i>Record</i> whose field names are intrinsic keys and whose values are objects	The intrinsic values used by code associated with this <i>realm</i>
[[GlobalObject]]	Object	The <i>global object</i> for this <i>realm</i>
[[GlobalEnv]]	<i>global Environment Record</i>	The global environment for this <i>realm</i>
[[TemplateMap]]	A <i>List</i> of <i>Record</i> { [[Site]]: <i>Parse Node</i> , [[Array]]: Object }.	<p>Template objects are canonicalized separately for each <i>realm</i> using its <i>Realm Record</i>'s [[TemplateMap]]. Each [[Site]] value is a <i>Parse Node</i> that is a <i>TemplateLiteral</i>. The associated [[Array]] value is the corresponding template object that is passed to a tag function.</p> <p>NOTE Once a <i>Parse Node</i> becomes unreachable, the corresponding [[Array]] is also unreachable, and it would be unobservable if an implementation removed the pair from the [[TemplateMap]] list.</p>
[[HostDefined]]	Any, default value is undefined .	Field reserved for use by hosts that need to associate additional information with a <i>Realm Record</i> .

9.2.1 CreateRealm ()

The abstract operation CreateRealm takes no arguments. It performs the following steps when called:

1. Let *realmRec* be a new *Realm Record*.
2. Perform *CreateIntrinsics*(*realmRec*).
3. Set *realmRec*.[[GlobalObject]] to **undefined**.
4. Set *realmRec*.[[GlobalEnv]] to **undefined**.
5. Set *realmRec*.[[TemplateMap]] to a new empty *List*.
6. Return *realmRec*.

9.2.2 CreateIntrinsics (*realmRec*)

The abstract operation CreateIntrinsics takes argument *realmRec*. It performs the following steps when called:

1. Let *intrinsic*s be a new Record.
2. Set *realmRec*.[[Intrinsics]] to *intrinsic*s.
3. Set fields of *intrinsic*s with the values listed in Table 8. 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 19 through 28. All object property values are newly created object values. All values that are built-in function objects are created by performing CreateBuiltinFunction(*steps*, *length*, *name*, *slots*, *realmRec*, *prototype*) where *steps* is the definition of that function provided by this specification, *name* is the initial value of the function's **name** property, *length* is the initial value of the function's **length** property, *slots* is a list of the names, if any, of the function's specified internal slots, and *prototype* is the specified value of the function's [[Prototype]] internal slot. The creation of the intrinsic and their properties must be ordered to avoid any dependencies upon objects that have not yet been created.
4. Perform AddRestrictedFunctionProperties(*intrinsic*s.[[%Function.prototype%]], *realmRec*).
5. Return *intrinsic*s.

9.2.3 SetRealmGlobalObject (*realmRec*, *globalObj*, *thisValue*)

The abstract operation SetRealmGlobalObject takes arguments *realmRec*, *globalObj*, and *thisValue*. It performs the following steps when called:

1. If *globalObj* is **undefined**, then
 - a. Let *intrinsic*s be *realmRec*.[[Intrinsics]].
 - b. Set *globalObj* to ! OrdinaryObjectCreate(*intrinsic*s.[[%Object.prototype%]]).
2. Assert: Type(*globalObj*) is Object.
3. If *thisValue* is **undefined**, set *thisValue* to *globalObj*.
4. Set *realmRec*.[[GlobalObject]] to *globalObj*.
5. Let *newGlobalEnv* be NewGlobalEnvironment(*globalObj*, *thisValue*).
6. Set *realmRec*.[[GlobalEnv]] to *newGlobalEnv*.
7. Return *realmRec*.

9.2.4 SetDefaultGlobalBindings (*realmRec*)

The abstract operation SetDefaultGlobalBindings takes argument *realmRec*. It performs the following steps when called:

1. Let *global* be *realmRec*.[[GlobalObject]].
2. For each property of the Global Object specified in clause 19, 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 listed in 19.2, 19.3, or 19.4 the value of the [[Value]] attribute is the corresponding intrinsic object from *realmRec*.
 - c. Perform ? DefinePropertyOrThrow(*global*, *name*, *desc*).
3. Return *global*.

9.3 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 per *agent* that is actually executing code. This is known as the *agent's running execution context*. All references to the *running execution context* in this specification denote the *running execution context* of the *surrounding agent*.

The *execution context 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 24](#).

Table 24: State Components for All Execution Contexts

Component	Purpose
code evaluation state	Any state needed to perform, suspend, and resume evaluation of the code associated with this <i>execution context</i> .
Function	If this <i>execution context</i> is evaluating the code of a <i>function object</i> , then the value of this component is that <i>function object</i> . If the context is evaluating the code of a <i>Script</i> or <i>Module</i> , the value is null .
Realm	The <i>Realm Record</i> from which associated code accesses ECMAScript resources.
ScriptOrModule	The <i>Module Record</i> or <i>Script Record</i> from which associated code originates. If there is no originating script or module, as is the case for the original <i>execution context</i> created in <i>InitializeHostDefinedRealm</i> , the value is null .

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 Record*. The value of the *Function* component of the *running execution context* is also called *the active function object*.

Execution contexts for ECMAScript code have the additional state components listed in [Table 25](#).

Table 25: Additional State Components for ECMAScript Code Execution Contexts

Component	Purpose
LexicalEnvironment	Identifies the Environment Record used to resolve identifier references made by code within this execution context .
VariableEnvironment	Identifies the Environment Record that holds bindings created by <i>VariableStatements</i> within this execution context .

The LexicalEnvironment and VariableEnvironment components of an execution context are always Environment Records.

Execution contexts representing the evaluation of generator objects have the additional state components listed in [Table 26](#).

Table 26: Additional State Components for Generator Execution Contexts

Component	Purpose
Generator	The generator object that this execution context is evaluating.

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.

9.3.1 GetActiveScriptOrModule ()

The abstract operation GetActiveScriptOrModule takes no arguments. It is used to determine the running script or module, based on the [running execution context](#). It performs the following steps when called:

1. If the [execution context stack](#) is empty, return **null**.
2. Let *ec* be the topmost [execution context](#) on the [execution context stack](#) whose ScriptOrModule component is not **null**.
3. If no such [execution context](#) exists, return **null**. Otherwise, return *ec*'s ScriptOrModule.

9.3.2 ResolveBinding (*name* [, *env*])

The abstract operation ResolveBinding takes argument *name* (a String) and optional argument *env* (an [Environment Record](#)). It is used to determine the binding of *name*. *env* can be used to explicitly provide the [Environment Record](#) that is to be searched for the binding. It performs the following steps when called:

1. If *env* is not present or if *env* is **undefined**, then
 - a. Set *env* to the [running execution context](#)'s LexicalEnvironment.
2. **Assert:** *env* is an [Environment Record](#).

3. If the code matching the syntactic production that is being evaluated is contained in *strict mode code*, let *strict* be **true**; else let *strict* be **false**.
4. Return ? *GetIdentifierReference*(*env*, *name*, *strict*).

NOTE The result of *ResolveBinding* is always a *Reference Record* whose *[[ReferencedName]]* field is *name*.

9.3.3 *GetThisEnvironment* ()

The abstract operation *GetThisEnvironment* takes no arguments. It finds the *Environment Record* that currently supplies the binding of the keyword **this**. It performs the following steps when called:

1. Let *env* be the *running execution context*'s *LexicalEnvironment*.
2. Repeat,
 - a. Let *exists* be *env*.*HasThisBinding*().
 - b. If *exists* is **true**, return *env*.
 - c. Let *outer* be *env*.*[[OuterEnv]]*.
 - d. **Assert**: *outer* is not **null**.
 - e. Set *env* to *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.

9.3.4 *ResolveThisBinding* ()

The abstract operation *ResolveThisBinding* takes no arguments. It determines the binding of the keyword **this** using the *LexicalEnvironment* of the *running execution context*. It performs the following steps when called:

1. Let *envRec* be *GetThisEnvironment*().
2. Return ? *envRec*.*GetThisBinding*().

9.3.5 *GetNewTarget* ()

The abstract operation *GetNewTarget* takes no arguments. It determines the *NewTarget* value using the *LexicalEnvironment* of the *running execution context*. It performs the following steps when called:

1. Let *envRec* be *GetThisEnvironment*().
2. **Assert**: *envRec* has a *[[NewTarget]]* field.
3. Return *envRec*.*[[NewTarget]]*.

9.3.6 *GetGlobalObject* ()

The abstract operation *GetGlobalObject* takes no arguments. It returns the *global object* used by the currently *running execution context*. It performs the following steps when called:

1. Let *currentRealm* be *the current Realm Record*.
2. Return *currentRealm*.*[[GlobalObject]]*.

9.4 Jobs and Host Operations to Enqueue Jobs

A *Job* is an [Abstract Closure](#) with no parameters that initiates an ECMAScript computation when no other ECMAScript computation is currently in progress.

Jobs are scheduled for execution by ECMAScript [host](#) environments. This specification describes the [host hook `HostEnqueuePromiseJob`](#) to schedule one kind of job; hosts may define additional [abstract operations](#) which schedule jobs. Such operations accept a [Job Abstract Closure](#) as the parameter and schedule it to be performed at some future time. Their implementations must conform to the following requirements:

- At some future point in time, when there is no [running execution context](#) and the [execution context stack](#) is empty, the implementation must:
 1. Perform any [host-defined](#) preparation steps.
 2. [Invoke](#) the [Job Abstract Closure](#).
 3. Perform any [host-defined](#) cleanup steps, after which the [execution context stack](#) must be empty.
- Only one [Job](#) may be actively undergoing evaluation at any point in time.
- Once evaluation of a [Job](#) starts, it must run to completion before evaluation of any other [Job](#) starts.
- The [Abstract Closure](#) must return a normal completion, implementing its own handling of errors.

NOTE 1 [Host](#) environments are not required to treat Jobs uniformly with respect to scheduling. For example, web browsers and Node.js treat Promise-handling Jobs as a higher priority than other work; future features may add Jobs that are not treated at such a high priority.

At any particular time, *scriptOrModule* (a [Script Record](#), a [Module Record](#), or **null**) is the *active script or module* if all of the following conditions are true:

- [GetActiveScriptOrModule\(\)](#) is *scriptOrModule*.
- If *scriptOrModule* is a [Script Record](#) or [Module Record](#), let *ec* be the topmost [execution context](#) on the [execution context stack](#) whose [ScriptOrModule](#) component is *scriptOrModule*. The [Realm](#) component of *ec* is *scriptOrModule*.[[[Realm](#)]].

At any particular time, an execution is *prepared to evaluate ECMAScript code* if all of the following conditions are true:

- The [execution context stack](#) is not empty.
- The [Realm](#) component of the topmost [execution context](#) on the [execution context stack](#) is a [Realm Record](#).

NOTE 2 [Host](#) environments may prepare an execution to evaluate code by pushing execution contexts onto the [execution context stack](#). The specific steps are [implementation-defined](#).

The specific choice of [Realm](#) is up to the [host environment](#). This initial [execution context](#) and [Realm](#) is only in use before any callback function is invoked. When a callback function related to a [Job](#), like a Promise handler, is invoked, the invocation pushes its own [execution context](#) and [Realm](#).

Particular kinds of Jobs have additional conformance requirements.

9.4.1 JobCallback Records

A *JobCallback Record* is a [Record](#) value used to store a [function object](#) and a [host-defined](#) value. Function objects that

are invoked via a `Job` enqueued by the `host` may have additional `host-defined` context. To propagate the state, `Job` Abstract Closures should not capture and call function objects directly. Instead, use `HostMakeJobCallback` and `HostCallJobCallback`.

NOTE The WHATWG HTML specification (<https://html.spec.whatwg.org/>), for example, uses the `host-defined` value to propagate the incumbent settings object for Promise callbacks.

`JobCallback` Records have the fields listed in [Table 27](#).

Table 27: `JobCallback` Record Fields

Field Name	Value	Meaning
[[Callback]]	A <code>function</code> object	The function to invoke when the <code>Job</code> is invoked.
[[HostDefined]]	Any, default value is <code>empty</code> .	Field reserved for use by hosts.

9.4.2 `HostMakeJobCallback` (*callback*)

The `host-defined` abstract operation `HostMakeJobCallback` takes argument *callback* (a `function` object).

The implementation of `HostMakeJobCallback` must conform to the following requirements:

- It must always complete normally (i.e., not return an `abrupt completion`).
- It must always return a `JobCallback` Record whose `[[Callback]]` field is *callback*.

The default implementation of `HostMakeJobCallback` performs the following steps when called:

1. **Assert:** `IsCallable(callback)` is `true`.
2. Return the `JobCallback` Record { `[[Callback]]`: *callback*, `[[HostDefined]]`: `empty` }.

ECMAScript hosts that are not web browsers must use the default implementation of `HostMakeJobCallback`.

NOTE This is called at the time that the callback is passed to the function that is responsible for its being eventually scheduled and run. For example, `promise.then(thenAction)` calls `MakeJobCallback` on `thenAction` at the time of invoking `Promise.prototype.then`, not at the time of scheduling the reaction `Job`.

9.4.3 `HostCallJobCallback` (*jobCallback*, *V*, *argumentsList*)

The `host-defined` abstract operation `HostCallJobCallback` takes arguments *jobCallback* (a `JobCallback` Record), *V* (an ECMAScript language value), and *argumentsList* (a `List` of ECMAScript language values).

The implementation of `HostCallJobCallback` must conform to the following requirements:

- It must always perform and return the result of `Call(jobCallback.[[Callback]], V, argumentsList)`.

NOTE This requirement means that hosts cannot change the `[[Call]]` behaviour of function objects defined in this specification.

The default implementation of `HostCallJobCallback` performs the following steps when called:

1. Assert: `IsCallable(jobCallback.[[Callback]])` is **true**.
2. Return ? `Call(jobCallback.[[Callback]], V, argumentsList)`.

ECMAScript hosts that are not web browsers must use the default implementation of `HostCallJobCallback`.

9.4.4 HostEnqueuePromiseJob (*job*, *realm*)

The `host-defined` abstract operation `HostEnqueuePromiseJob` takes arguments *job* (a `Job Abstract Closure`) and *realm* (a `Realm Record` or **null**). It schedules *job* to be performed at some future time. The Abstract Closures used with this algorithm are intended to be related to the handling of Promises, or otherwise, to be scheduled with equal priority to Promise handling operations.

The implementation of `HostEnqueuePromiseJob` must conform to the requirements in 9.4 as well as the following:

- If *realm* is not **null**, each time *job* is invoked the implementation must perform `implementation-defined` steps such that execution is `prepared to evaluate ECMAScript code` at the time of *job*'s invocation.
- Let *scriptOrModule* be `GetActiveScriptOrModule()` at the time `HostEnqueuePromiseJob` is invoked. If *realm* is not **null**, each time *job* is invoked the implementation must perform `implementation-defined` steps such that *scriptOrModule* is the `active script or module` at the time of *job*'s invocation.
- Jobs must run in the same order as the `HostEnqueuePromiseJob` invocations that scheduled them.

NOTE

The *realm* for Jobs returned by `NewPromiseResolveThenableJob` is usually the result of calling `GetFunctionRealm` on the *then function object*. The *realm* for Jobs returned by `NewPromiseReactionJob` is usually the result of calling `GetFunctionRealm` on the handler if the handler is not **undefined**. If the handler is **undefined**, *realm* is **null**. For both kinds of Jobs, when `GetFunctionRealm` completes abnormally (i.e. called on a revoked Proxy), *realm* is the current `Realm` at the time of the `GetFunctionRealm` call. When the *realm* is **null**, no user ECMAScript code will be evaluated and no new ECMAScript objects (e.g. Error objects) will be created. The WHATWG HTML specification (<https://html.spec.whatwg.org/>), for example, uses *realm* to check for the ability to run script and for the `entry` concept.

9.5 InitializeHostDefinedRealm ()

The abstract operation `InitializeHostDefinedRealm` takes no arguments. It performs the following steps when called:

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. Set the `ScriptOrModule` of *newContext* to **null**.
6. Push *newContext* onto the `execution context stack`; *newContext* is now the `running execution context`.
7. If the *host* requires use of an `exotic object` to serve as *realm*'s `global object`, let *global* be such an object created in a `host-defined` manner. Otherwise, let *global* be **undefined**, indicating that an `ordinary object` should be created as the `global object`.
8. If the *host* requires that the **this** binding in *realm*'s global scope return an object other than the `global object`, let *thisValue* be such an object created in a `host-defined` manner. Otherwise, let *thisValue* be **undefined**,

indicating that *realm*'s global **this** binding should be the *global object*.

9. Perform `SetRealmGlobalObject(realm, global, thisValue)`.
10. Let *globalObj* be ? `SetDefaultGlobalBindings(realm)`.
11. Create any *host-defined global object* properties on *globalObj*.
12. Return `NormalCompletion(empty)`.

9.6 Agents

An *agent* comprises a set of ECMAScript execution contexts, an *execution context stack*, a *running execution context*, an *Agent Record*, and an *executing thread*. Except for the *executing thread*, the constituents of an *agent* belong exclusively to that *agent*.

An *agent's executing thread* executes a job on the *agent's* execution contexts independently of other agents, except that an *executing thread* may be used as the *executing thread* by multiple agents, provided none of the agents sharing the thread have an *Agent Record* whose `[[CanBlock]]` property is **true**.

NOTE 1 Some web browsers share a single *executing thread* across multiple unrelated tabs of a browser window, for example.

While an *agent's executing thread* executes jobs, the *agent* is the *surrounding agent* for the code in those jobs. The code uses the *surrounding agent* to access the specification level execution objects held within the *agent*: the *running execution context*, the *execution context stack*, and the *Agent Record's* fields.

Table 28: Agent Record Fields

Field Name	Value	Meaning
[[LittleEndian]]	Boolean	The default value computed for the <i>isLittleEndian</i> parameter when it is needed by the algorithms GetValueFromBuffer and SetValueInBuffer . The choice is implementation-defined and should be the alternative that is most efficient for the implementation. Once the value has been observed it cannot change.
[[CanBlock]]	Boolean	Determines whether the agent can block or not.
[[Signifier]]	Any globally-unique value	Uniquely identifies the agent within its agent cluster .
[[IsLockFree1]]	Boolean	true if atomic operations on one-byte values are lock-free, false otherwise.
[[IsLockFree2]]	Boolean	true if atomic operations on two-byte values are lock-free, false otherwise.
[[IsLockFree8]]	Boolean	true if atomic operations on eight-byte values are lock-free, false otherwise.
[[CandidateExecution]]	A candidate execution Record	See the memory model .
[[KeptAlive]]	List of objects	Initially a new empty List , representing the list of objects to be kept alive until the end of the current Job

Once the values of [[Signifier]], [[IsLockFree1]], and [[IsLockFree2]] have been observed by any [agent](#) in the [agent cluster](#) they cannot change.

NOTE 2

The values of [[IsLockFree1]] and [[IsLockFree2]] are not necessarily determined by the hardware, but may also reflect implementation choices that can vary over time and between ECMAScript implementations.

There is no [[IsLockFree4]] property: 4-byte atomic operations are always lock-free.

In practice, if an atomic operation is implemented with any type of lock the operation is not lock-free. Lock-free does not imply wait-free: there is no upper bound on how many machine steps may be required to complete a lock-free atomic operation.

That an atomic access of size n is lock-free does not imply anything about the (perceived) atomicity of non-atomic accesses of size n , specifically, non-atomic accesses may still be performed as a sequence of several separate memory accesses. See [ReadSharedMemory](#) and [WriteSharedMemory](#) for details.

NOTE 3 An **agent** is a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation.

9.6.1 AgentSignifier ()

The abstract operation AgentSignifier takes no arguments. It performs the following steps when called:

1. Let *AR* be the **Agent Record** of the **surrounding agent**.
2. Return *AR*.[[Signifier]].

9.6.2 AgentCanSuspend ()

The abstract operation AgentCanSuspend takes no arguments. It performs the following steps when called:

1. Let *AR* be the **Agent Record** of the **surrounding agent**.
2. Return *AR*.[[CanBlock]].

NOTE In some environments it may not be reasonable for a given **agent** to suspend. For example, in a web browser environment, it may be reasonable to disallow suspending a document's main event handling thread, while still allowing workers' event handling threads to suspend.

9.7 Agent Clusters

An *agent cluster* is a maximal set of agents that can communicate by operating on shared memory.

NOTE 1 Programs within different agents may share memory by unspecified means. At a minimum, the backing memory for SharedArrayBuffer objects can be shared among the agents in the cluster.

There may be agents that can communicate by message passing that cannot share memory; they are never in the same agent cluster.

Every **agent** belongs to exactly one agent cluster.

NOTE 2 The agents in a cluster need not all be alive at some particular point in time. If **agent A** creates another **agent B**, after which **A** terminates and **B** creates **agent C**, the three agents are in the same cluster if **A** could share some memory with **B** and **B** could share some memory with **C**.

All agents within a cluster must have the same value for the [[LittleEndian]] property in their respective **Agent Records**.

NOTE 3 If different agents within an agent cluster have different values of [[LittleEndian]] it becomes hard to use shared memory for multi-byte data.

All agents within a cluster must have the same values for the [[IsLockFree1]] property in their respective **Agent Records**; similarly for the [[IsLockFree2]] property.

All agents within a cluster must have different values for the `[[Signifier]]` property in their respective [Agent Records](#).

An embedding may deactivate (stop forward progress) or activate (resume forward progress) an [agent](#) without the [agent's](#) knowledge or cooperation. If the embedding does so, it must not leave some agents in the cluster active while other agents in the cluster are deactivated indefinitely.

NOTE 4 The purpose of the preceding restriction is to avoid a situation where an [agent](#) deadlocks or starves because another [agent](#) has been deactivated. For example, if an HTML shared worker that has a lifetime independent of documents in any windows were allowed to share memory with the dedicated worker of such an independent document, and the document and its dedicated worker were to be deactivated while the dedicated worker holds a lock (say, the document is pushed into its window's history), and the shared worker then tries to acquire the lock, then the shared worker will be blocked until the dedicated worker is activated again, if ever. Meanwhile other workers trying to access the shared worker from other windows will starve.

The implication of the restriction is that it will not be possible to share memory between agents that don't belong to the same suspend/wake collective within the embedding.

An embedding may terminate an [agent](#) without any of the [agent's](#) cluster's other agents' prior knowledge or cooperation. If an [agent](#) is terminated not by programmatic action of its own or of another [agent](#) in the cluster but by forces external to the cluster, then the embedding must choose one of two strategies: Either terminate all the agents in the cluster, or provide reliable APIs that allow the agents in the cluster to coordinate so that at least one remaining member of the cluster will be able to detect the termination, with the termination data containing enough information to identify the [agent](#) that was terminated.

NOTE 5 Examples of that type of termination are: operating systems or users terminating agents that are running in separate processes; the embedding itself terminating an [agent](#) that is running in-process with the other agents when per-[agent](#) resource accounting indicates that the [agent](#) is runaway.

Prior to any evaluation of any ECMAScript code by any [agent](#) in a cluster, the `[[CandidateExecution]]` field of the [Agent Record](#) for all agents in the cluster is set to the initial [candidate execution](#). The initial [candidate execution](#) is an [empty candidate execution](#) whose `[[EventsRecords]]` field is a [List](#) containing, for each [agent](#), an [Agent Events Record](#) whose `[[AgentSignifier]]` field is that [agent's](#) signifier, and whose `[[EventList]]` and `[[AgentSynchronizesWith]]` fields are empty Lists.

NOTE 6 All agents in an agent cluster share the same [candidate execution](#) in its [Agent Record's](#) `[[CandidateExecution]]` field. The [candidate execution](#) is a specification mechanism used by the [memory model](#).

NOTE 7 An agent cluster is a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation.

9.8 Forward Progress

For an [agent](#) to *make forward progress* is for it to perform an evaluation step according to this specification.

An [agent](#) becomes *blocked* when its [running execution context](#) waits synchronously and indefinitely for an external event. Only agents whose [Agent Record](#)'s `[[CanBlock]]` property is **true** can become blocked in this sense. An *unblocked agent* is one that is not blocked.

Implementations must ensure that:

- every unblocked [agent](#) with a dedicated [executing thread](#) eventually makes forward progress
- in a set of agents that share an [executing thread](#), one [agent](#) eventually makes forward progress
- an [agent](#) does not cause another [agent](#) to become blocked except via explicit APIs that provide blocking.

NOTE This, along with the liveness guarantee in the [memory model](#), ensures that all `SeqCst` writes eventually become observable to all agents.

9.9 Processing Model of WeakRef and FinalizationRegistry Objects

9.9.1 Objectives

This specification does not make any guarantees that any object will be garbage collected. Objects which are not [live](#) may be released after long periods of time, or never at all. For this reason, this specification uses the term "may" when describing behaviour triggered by garbage collection.

The semantics of [WeakRef](#) and [FinalizationRegistry](#) objects is based on two operations which happen at particular points in time:

- When **`WeakRef.prototype.deref`** is called, the referent (if **`undefined`** is not returned) is kept alive so that subsequent, synchronous accesses also return the object. This list is reset when synchronous work is done using the [ClearKeptObjects](#) abstract operation.
- When an object which is registered with a [FinalizationRegistry](#) becomes unreachable, a call of the [FinalizationRegistry](#)'s cleanup callback may eventually be made, after synchronous ECMAScript execution completes. The [FinalizationRegistry](#) cleanup is performed with the [CleanupFinalizationRegistry](#) abstract operation.

Neither of these actions ([ClearKeptObjects](#) or [CleanupFinalizationRegistry](#)) may interrupt synchronous ECMAScript execution. Because hosts may assemble longer, synchronous ECMAScript execution runs, this specification defers the scheduling of [ClearKeptObjects](#) and [CleanupFinalizationRegistry](#) to the [host environment](#).

Some ECMAScript implementations include garbage collector implementations which run in the background, including when ECMAScript is idle. Letting the [host environment](#) schedule [CleanupFinalizationRegistry](#) allows it to resume ECMAScript execution in order to run finalizer work, which may free up held values, reducing overall memory usage.

9.9.2 Liveness

For some set of objects *S*, a *hypothetical WeakRef-oblivious* execution with respect to *S* is an execution whereby the abstract operation [WeakRefDeref](#) of a [WeakRef](#) whose referent is an element of *S* always returns **`undefined`**.

NOTE 1 [WeakRef](#)-obliviousness, together with liveness, capture two notions. One, that a [WeakRef](#) itself does not keep an object alive. Two, that cycles in liveness does not imply that an object is live. To be concrete, if determining *obj*'s liveness depends on determining the liveness of another [WeakRef](#) referent, *obj2*, *obj2*'s liveness cannot assume *obj*'s liveness, which would be circular reasoning.

NOTE 2 [WeakRef](#)-obliviousness is defined on sets of objects instead of individual objects to account for cycles. If it were defined on individual objects, then an object in a cycle will be considered live even though its Object value is only observed via WeakRefs of other objects in the cycle.

NOTE 3 Colloquially, we say that an individual object is live if every set of objects containing it is live.

At any point during evaluation, a set of objects *S* is considered *live* if either of the following conditions is met:

- Any element in *S* is included in any [agent](#)'s `[[KeptAlive]]` [List](#).
- There exists a valid future hypothetical [WeakRef](#)-oblivious execution with respect to *S* that observes the Object value of any object in *S*.

NOTE 4 The intuition the second condition above intends to capture is that an object is live if its identity is observable via non-[WeakRef](#) means. An object's identity may be observed by observing a strict equality comparison between objects or observing the object being used as key in a Map.

NOTE 5 Presence of an object in a field, an internal slot, or a property does not imply that the object is live. For example if the object in question is never passed back to the program, then it cannot be observed.

This is the case for keys in a [WeakMap](#), members of a [WeakSet](#), as well as the `[[WeakRefTarget]]` and `[[UnregisterToken]]` fields of a [FinalizationRegistry](#) Cell record.

The above definition implies that, if a key in a [WeakMap](#) is not live, then its corresponding value is not necessarily live either.

NOTE 6 Liveness is the lower bound for guaranteeing which [WeakRefs](#) engines must not empty. Liveness as defined here is undecidable. In practice, engines use conservative approximations such as reachability. There is expected to be significant implementation leeway.

9.9.3 Execution

At any time, if a set of objects *S* is not *live*, an ECMAScript implementation may perform the following steps atomically:

1. For each element *obj* of *S*, do
 - a. For each [WeakRef](#) *ref* such that *ref*.`[[WeakRefTarget]]` is *obj*, do
 - i. Set *ref*.`[[WeakRefTarget]]` to empty.

- b. For each `FinalizationRegistry` *fg* such that *fg*.[[Cells]] contains a `Record` *cell* such that *cell*.[[WeakRefTarget]] is *obj*, do
 - i. Set *cell*.[[WeakRefTarget]] to empty.
 - ii. Optionally, perform !`HostEnqueueFinalizationRegistryCleanupJob`(*fg*).
- c. For each `WeakMap` *map* such that *map*.[[WeakMapData]] contains a `Record` *r* such that *r*.[[Key]] is *obj*, do
 - i. Set *r*.[[Key]] to empty.
 - ii. Set *r*.[[Value]] to empty.
- d. For each `WeakSet` *set* such that *set*.[[WeakSetData]] contains *obj*, do
 - i. Replace the element of *set*.[[WeakSetData]] whose value is *obj* with an element whose value is empty.

NOTE 1

Together with the definition of liveness, this clause prescribes legal optimizations that an implementation may apply regarding WeakRefs.

It is possible to access an object without observing its identity. Optimizations such as dead variable elimination and scalar replacement on properties of non-escaping objects whose identity is not observed are allowed. These optimizations are thus allowed to observably empty WeakRefs that point to such objects.

On the other hand, if an object's identity is observable, and that object is in the [[WeakRefTarget]] internal slot of a `WeakRef`, optimizations such as rematerialization that observably empty the `WeakRef` are prohibited.

Because calling `HostEnqueueFinalizationRegistryCleanupJob` is optional, registered objects in a `FinalizationRegistry` do not necessarily hold that `FinalizationRegistry` *live*. Implementations may omit `FinalizationRegistry` callbacks for any reason, e.g., if the `FinalizationRegistry` itself becomes dead, or if the application is shutting down.

NOTE 2

Implementations are not obligated to empty WeakRefs for maximal sets of non-*live* objects.

If an implementation chooses a non-*live* set *S* in which to empty WeakRefs, it must empty WeakRefs for all objects in *S* simultaneously. In other words, an implementation must not empty a `WeakRef` pointing to an object *obj* without emptying out other WeakRefs that, if not emptied, could result in an execution that observes the Object value of *obj*.

9.9.4 Host Hooks

9.9.4.1 `HostEnqueueFinalizationRegistryCleanupJob` (*finalizationRegistry*)

The abstract operation `HostEnqueueFinalizationRegistryCleanupJob` takes argument *finalizationRegistry* (a `FinalizationRegistry`). `HostEnqueueFinalizationRegistryCleanupJob` is an **implementation-defined** abstract operation that is expected to call `CleanupFinalizationRegistry`(*finalizationRegistry*) at some point in the future, if possible. The *host*'s responsibility is to make this call at a time which does not interrupt synchronous ECMAScript code execution.

9.10 `ClearKeptObjects` ()

The abstract operation `ClearKeptObjects` takes no arguments. ECMAScript implementations are expected to call `ClearKeptObjects` when a synchronous sequence of ECMAScript executions completes. It performs the following steps when called:

1. Let *agentRecord* be the [surrounding agent's Agent Record](#).
2. Set *agentRecord*.[[KeptAlive]] to a new empty [List](#).

9.11 AddToKeptObjects (*object*)

The abstract operation `AddToKeptObjects` takes argument *object* (an Object). It performs the following steps when called:

1. Let *agentRecord* be the [surrounding agent's Agent Record](#).
2. Append *object* to *agentRecord*.[[KeptAlive]].

NOTE

When the abstract operation `AddToKeptObjects` is called with a target object reference, it adds the target to a list that will point strongly at the target until `ClearKeptObjects` is called.

9.12 CleanupFinalizationRegistry (*finalizationRegistry*)

The abstract operation `CleanupFinalizationRegistry` takes argument *finalizationRegistry* (a [FinalizationRegistry](#)). It performs the following steps when called:

1. [Assert](#): *finalizationRegistry* has [[Cells]] and [[CleanupCallback]] internal slots.
2. Let *callback* be *finalizationRegistry*.[[CleanupCallback]].
3. While *finalizationRegistry*.[[Cells]] contains a [Record cell](#) such that *cell*.[[WeakRefTarget]] is empty, an implementation may perform the following steps:
 - a. Choose any such *cell*.
 - b. Remove *cell* from *finalizationRegistry*.[[Cells]].
 - c. Perform ? [Call](#)(*callback*, `undefined`, « *cell*.[[HeldValue]] »).
4. Return [NormalCompletion](#)(`undefined`).

10 Ordinary and Exotic Objects Behaviours

10.1 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 (and 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 which is used to fulfill the extensibility-related internal method invariants specified in [6.1.7.3](#). Namely, once the value of an object's [[Extensible]] internal slot has been set to `false`, it is no longer possible to add properties to the object, to modify the value of the object's

[[Prototype]] internal slot, or to subsequently change the value of [[Extensible]] 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.

Each [ordinary object](#) internal method delegates to a similarly-named abstract operation. If such an abstract operation depends on another internal method, then the internal method is invoked on *O* rather than calling the similarly-named abstract operation directly. These semantics ensure that exotic objects have their overridden internal methods invoked when [ordinary object](#) internal methods are applied to them.

10.1.1 [[GetPrototypeOf]] ()

The [[GetPrototypeOf]] internal method of an [ordinary object](#) *O* takes no arguments. It performs the following steps when called:

1. Return ! [OrdinaryGetPrototypeOf](#)(*O*).

10.1.1.1 OrdinaryGetPrototypeOf (*O*)

The abstract operation [OrdinaryGetPrototypeOf](#) takes argument *O* (an Object). It performs the following steps when called:

1. Return *O*.[[Prototype]].

10.1.2 [[SetPrototypeOf]] (*V*)

The [[SetPrototypeOf]] internal method of an [ordinary object](#) *O* takes argument *V* (an Object or **null**). It performs the following steps when called:

1. Return ! [OrdinarySetPrototypeOf](#)(*O*, *V*).

10.1.2.1 OrdinarySetPrototypeOf (*O*, *V*)

The abstract operation [OrdinarySetPrototypeOf](#) takes arguments *O* (an Object) and *V* (an [ECMAScript language value](#)). It performs the following steps when called:

1. **Assert**: Either [Type](#)(*V*) is Object or [Type](#)(*V*) is Null.
2. Let *current* be *O*.[[Prototype]].
3. If [SameValue](#)(*V*, *current*) is **true**, return **true**.
4. Let *extensible* be *O*.[[Extensible]].
5. If *extensible* is **false**, return **false**.
6. Let *p* be *V*.
7. Let *done* be **false**.
8. Repeat, while *done* is **false**,
 - a. If *p* is **null**, set *done* to **true**.
 - b. Else if [SameValue](#)(*p*, *O*) is **true**, return **false**.
 - c. Else,
 - i. If *p*.[[GetPrototypeOf]] is not the [ordinary object](#) internal method defined in 10.1.1, set *done* to **true**.
 - ii. Else, set *p* to *p*.[[Prototype]].

9. Set $O.[\text{Prototype}]$ to V .

10. Return **true**.

NOTE

The loop in step 8 guarantees that there will be no circularities in any prototype chain that only includes objects that use the [ordinary object](#) definitions for `[[GetPrototypeOf]]` and `[[SetPrototypeOf]]`.

10.1.3 `[[IsExtensible]]` ()

The `[[IsExtensible]]` internal method of an [ordinary object](#) O takes no arguments. It performs the following steps when called:

1. Return ! `OrdinaryIsExtensible(O)`.

10.1.3.1 `OrdinaryIsExtensible` (O)

The abstract operation `OrdinaryIsExtensible` takes argument O (an Object). It performs the following steps when called:

1. Return $O.[\text{Extensible}]$.

10.1.4 `[[PreventExtensions]]` ()

The `[[PreventExtensions]]` internal method of an [ordinary object](#) O takes no arguments. It performs the following steps when called:

1. Return ! `OrdinaryPreventExtensions(O)`.

10.1.4.1 `OrdinaryPreventExtensions` (O)

The abstract operation `OrdinaryPreventExtensions` takes argument O (an Object). It performs the following steps when called:

1. Set $O.[\text{Extensible}]$ to **false**.
2. Return **true**.

10.1.5 `[[GetOwnProperty]]` (P)

The `[[GetOwnProperty]]` internal method of an [ordinary object](#) O takes argument P (a property key). It performs the following steps when called:

1. Return ! `OrdinaryGetOwnProperty(O, P)`.

10.1.5.1 `OrdinaryGetOwnProperty` (O, P)

The abstract operation `OrdinaryGetOwnProperty` takes arguments O (an Object) and P (a property key). It performs the following steps when called:

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,
 - a. **Assert**: X is an **accessor property**.
 - b. Set $D.[[Get]]$ to the value of X 's $[[Get]]$ attribute.
 - c. 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 .

10.1.6 **[[DefineOwnProperty]]** (P , $Desc$)

The **[[DefineOwnProperty]]** internal method of an **ordinary object** O takes arguments P (a property key) and $Desc$ (a **Property Descriptor**). It performs the following steps when called:

1. Return ? **OrdinaryDefineOwnProperty**(O , P , $Desc$).

10.1.6.1 **OrdinaryDefineOwnProperty** (O , P , $Desc$)

The abstract operation **OrdinaryDefineOwnProperty** takes arguments O (an Object), P (a property key), and $Desc$ (a **Property Descriptor**). It performs the following steps when called:

1. Let $current$ be ? $O.[[GetOwnProperty]](P)$.
2. Let $extensible$ be ? **IsExtensible**(O).
3. Return **ValidateAndApplyPropertyDescriptor**(O , P , $extensible$, $Desc$, $current$).

10.1.6.2 **IsCompatiblePropertyDescriptor** ($Extensible$, $Desc$, $Current$)

The abstract operation **IsCompatiblePropertyDescriptor** takes arguments $Extensible$ (a Boolean), $Desc$ (a **Property Descriptor**), and $Current$ (a **Property Descriptor**). It performs the following steps when called:

1. Return **ValidateAndApplyPropertyDescriptor**(**undefined**, **undefined**, $Extensible$, $Desc$, $Current$).

10.1.6.3 **ValidateAndApplyPropertyDescriptor** (O , P , $extensible$, $Desc$, $current$)

The abstract operation **ValidateAndApplyPropertyDescriptor** takes arguments O (an Object or **undefined**), P (a property key), $extensible$ (a Boolean), $Desc$ (a **Property Descriptor**), and $current$ (a **Property Descriptor**). It performs the following steps when called:

NOTE If **undefined** is passed as O , only validation is performed and no object updates are performed.

1. **Assert**: If O is not **undefined**, then **IsPropertyKey**(P) is **true**.
2. If $current$ is **undefined**, then
 - a. If $extensible$ is **false**, return **false**.

- b. **Assert:** *extensible* is **true**.
 - c. If **IsGenericDescriptor**(*Desc*) is **true** or **IsDataDescriptor**(*Desc*) is **true**, then
 - i. If *O* is not **undefined**, create an own **data property** named *P* of object *O* whose **[[Value]]**, **[[Writable]]**, **[[Enumerable]]**, and **[[Configurable]]** attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its **default value**.
 - d. Else,
 - i. **Assert:** **! IsAccessorDescriptor**(*Desc*) is **true**.
 - ii. If *O* is not **undefined**, create an own **accessor property** named *P* of object *O* whose **[[Get]]**, **[[Set]]**, **[[Enumerable]]**, and **[[Configurable]]** attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its **default value**.
 - e. Return **true**.
3. If every field in *Desc* is absent, return **true**.
 4. If *current*.**[[Configurable]]** is **false**, then
 - a. If *Desc*.**[[Configurable]]** is present and its value is **true**, return **false**.
 - b. If *Desc*.**[[Enumerable]]** is present and **! SameValue**(*Desc*.**[[Enumerable]]**, *current*.**[[Enumerable]]**) is **false**, return **false**.
 5. If **! IsGenericDescriptor**(*Desc*) is **true**, then
 - a. NOTE: No further validation is required.
 6. Else if **! SameValue**(**! IsDataDescriptor**(*current*), **! IsDataDescriptor**(*Desc*)) is **false**, then
 - a. If *current*.**[[Configurable]]** is **false**, return **false**.
 - b. If **IsDataDescriptor**(*current*) is **true**, then
 - i. If *O* is not **undefined**, convert the property named *P* of object *O* from a **data property** to an **accessor property**. Preserve the existing values of the converted property's **[[Configurable]]** and **[[Enumerable]]** attributes and set the rest of the property's attributes to their **default values**.
 - c. Else,
 - i. If *O* is not **undefined**, 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**.
 7. Else if **IsDataDescriptor**(*current*) and **IsDataDescriptor**(*Desc*) are both **true**, then
 - a. If *current*.**[[Configurable]]** is **false** and *current*.**[[Writable]]** is **false**, then
 - i. If *Desc*.**[[Writable]]** is present and *Desc*.**[[Writable]]** is **true**, return **false**.
 - ii. If *Desc*.**[[Value]]** is present and **SameValue**(*Desc*.**[[Value]]**, *current*.**[[Value]]**) is **false**, return **false**.
 - iii. Return **true**.
 8. Else,
 - a. **Assert:** **! IsAccessorDescriptor**(*current*) and **! IsAccessorDescriptor**(*Desc*) are both **true**.
 - b. If *current*.**[[Configurable]]** is **false**, then
 - i. If *Desc*.**[[Set]]** is present and **SameValue**(*Desc*.**[[Set]]**, *current*.**[[Set]]**) is **false**, return **false**.
 - ii. If *Desc*.**[[Get]]** is present and **SameValue**(*Desc*.**[[Get]]**, *current*.**[[Get]]**) is **false**, return **false**.
 - iii. Return **true**.
 9. If *O* is not **undefined**, then
 - a. For each field of *Desc* that is present, set the corresponding attribute of the property named *P* of object *O* to the value of the field.
 10. Return **true**.

10.1.7 **[[HasProperty]]** (*P*)

The `[[HasProperty]]` internal method of an [ordinary object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. Return ? `OrdinaryHasProperty(O, P)`.

10.1.7.1 OrdinaryHasProperty (*O*, *P*)

The abstract operation `OrdinaryHasProperty` takes arguments *O* (an Object) and *P* (a property key). It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *hasOwn* be ? *O*.`[[GetOwnProperty]](P)`.
3. If *hasOwn* is not **undefined**, return **true**.
4. Let *parent* be ? *O*.`[[GetPrototypeOf]]()`.
5. If *parent* is not **null**, then
 - a. Return ? *parent*.`[[HasProperty]](P)`.
6. Return **false**.

10.1.8 `[[Get]]` (*P*, *Receiver*)

The `[[Get]]` internal method of an [ordinary object](#) *O* takes arguments *P* (a property key) and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. Return ? `OrdinaryGet(O, P, Receiver)`.

10.1.8.1 OrdinaryGet (*O*, *P*, *Receiver*)

The abstract operation `OrdinaryGet` takes arguments *O* (an Object), *P* (a property key), and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *desc* be ? *O*.`[[GetOwnProperty]](P)`.
3. If *desc* is **undefined**, then
 - a. Let *parent* be ? *O*.`[[GetPrototypeOf]]()`.
 - b. If *parent* is **null**, return **undefined**.
 - c. Return ? *parent*.`[[Get]](P, Receiver)`.
4. If `IsDataDescriptor(desc)` is **true**, return *desc*.`[[Value]]`.
5. **Assert:** `IsAccessorDescriptor(desc)` is **true**.
6. Let *getter* be *desc*.`[[Get]]`.
7. If *getter* is **undefined**, return **undefined**.
8. Return ? `Call(getter, Receiver)`.

10.1.9 `[[Set]]` (*P*, *V*, *Receiver*)

The `[[Set]]` internal method of an [ordinary object](#) *O* takes arguments *P* (a property key), *V* (an [ECMAScript language value](#)), and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. Return ? `OrdinarySet(O, P, V, Receiver)`.

10.1.9.1 OrdinarySet (*O*, *P*, *V*, *Receiver*)

The abstract operation OrdinarySet takes arguments *O* (an Object), *P* (a property key), *V* (an ECMAScript language value), and *Receiver* (an ECMAScript language value). It performs the following steps when called:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *ownDesc* be ? *O*.[[GetOwnProperty]](*P*).
3. Return OrdinarySetWithOwnDescriptor(*O*, *P*, *V*, *Receiver*, *ownDesc*).

10.1.9.2 OrdinarySetWithOwnDescriptor (*O*, *P*, *V*, *Receiver*, *ownDesc*)

The abstract operation OrdinarySetWithOwnDescriptor takes arguments *O* (an Object), *P* (a property key), *V* (an ECMAScript language value), *Receiver* (an ECMAScript language value), and *ownDesc* (a Property Descriptor or **undefined**). It performs the following steps when called:

1. Assert: IsPropertyKey(*P*) is **true**.
2. If *ownDesc* is **undefined**, then
 - a. Let *parent* be ? *O*.[[GetPrototypeOf]]().
 - b. If *parent* is not **null**, then
 - i. Return ? *parent*.[[Set]](*P*, *V*, *Receiver*).
 - c. Else,
 - i. Set *ownDesc* to the PropertyDescriptor { [[Value]]: **undefined**, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true** }.
3. 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 ? *Receiver*.[[GetOwnProperty]](*P*).
 - d. If *existingDescriptor* is not **undefined**, then
 - i. If IsAccessorDescriptor(*existingDescriptor*) is **true**, return **false**.
 - ii. If *existingDescriptor*.[[Writable]] is **false**, return **false**.
 - iii. Let *valueDesc* be the PropertyDescriptor { [[Value]]: *V* }.
 - iv. Return ? *Receiver*.[[DefineOwnProperty]](*P*, *valueDesc*).
 - e. Else,
 - i. Assert: *Receiver* does not currently have a property *P*.
 - ii. Return ? CreateDataProperty(*Receiver*, *P*, *V*).
4. Assert: IsAccessorDescriptor(*ownDesc*) is **true**.
5. Let *setter* be *ownDesc*.[[Set]].
6. If *setter* is **undefined**, return **false**.
7. Perform ? Call(*setter*, *Receiver*, « *V* »).
8. Return **true**.

10.1.10 [[Delete]] (*P*)

The [[Delete]] internal method of an ordinary object *O* takes argument *P* (a property key). It performs the following steps when called:

1. Return ? OrdinaryDelete(*O*, *P*).

10.1.10.1 OrdinaryDelete (*O*, *P*)

The abstract operation `OrdinaryDelete` takes arguments *O* (an Object) and *P* (a property key). It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *desc* be ? *O*.[`[[GetOwnProperty]]`](*P*).
3. If *desc* is **undefined**, return **true**.
4. If *desc*.[`[[Configurable]]`] is **true**, then
 - a. Remove the own property with name *P* from *O*.
 - b. Return **true**.
5. Return **false**.

10.1.11 `[[OwnPropertyKeys]]` ()

The `[[OwnPropertyKeys]]` internal method of an **ordinary object** *O* takes no arguments. It performs the following steps when called:

1. Return ! `OrdinaryOwnPropertyKeys(O)`.

10.1.11.1 `OrdinaryOwnPropertyKeys` (*O*)

The abstract operation `OrdinaryOwnPropertyKeys` takes argument *O* (an Object). It performs the following steps when called:

1. Let *keys* be a new empty **List**.
2. For each own property key *P* of *O* such that *P* is an **array index**, in ascending numeric index order, do
 - a. Add *P* as the last element of *keys*.
3. For each own property key *P* of *O* such that `Type(P)` is String and *P* is not an **array index**, in ascending chronological order of property creation, do
 - a. Add *P* as the last element of *keys*.
4. For each own property key *P* of *O* such that `Type(P)` is Symbol, in ascending chronological order of property creation, do
 - a. Add *P* as the last element of *keys*.
5. Return *keys*.

10.1.12 `OrdinaryObjectCreate` (*proto* [, *additionalInternalSlotsList*])

The abstract operation `OrdinaryObjectCreate` takes argument *proto* (an Object or **null**) and optional argument *additionalInternalSlotsList* (a **List** of names of internal slots). It is used to specify the runtime creation of new ordinary objects. *additionalInternalSlotsList* contains the names of additional internal slots that must be defined as part of the object, beyond `[[Prototype]]` and `[[Extensible]]`. If *additionalInternalSlotsList* is not provided, a new empty **List** is used. It performs the following steps when called:

1. Let *internalSlotsList* be « `[[Prototype]]`, `[[Extensible]]` ».
2. If *additionalInternalSlotsList* is present, append each of its elements to *internalSlotsList*.
3. Let *O* be ! `MakeBasicObject(internalSlotsList)`.
4. Set *O*.[`[[Prototype]]`] to *proto*.
5. Return *O*.

NOTE

Although `OrdinaryObjectCreate` does little more than call `MakeBasicObject`, its use communicates the intention to create an [ordinary object](#), and not an exotic one. Thus, within this specification, it is not called by any algorithm that subsequently modifies the internal methods of the object in ways that would make the result non-ordinary. Operations that create exotic objects invoke `MakeBasicObject` directly.

10.1.13 OrdinaryCreateFromConstructor (*constructor*, *intrinsicDefaultProto* [, *internalSlotsList*])

The abstract operation `OrdinaryCreateFromConstructor` takes arguments *constructor* and *intrinsicDefaultProto* and optional argument *internalSlotsList* (a [List](#) of names of internal slots). It creates an [ordinary object](#) whose `[[Prototype]]` value is retrieved from a *constructor*'s "**prototype**" property, if it exists. Otherwise the intrinsic named by *intrinsicDefaultProto* is used for `[[Prototype]]`. *internalSlotsList* contains the names of additional internal slots that must be defined as part of the object. If *internalSlotsList* is not provided, a new empty [List](#) is used. It performs the following steps when called:

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. Return ! `OrdinaryObjectCreate(proto, internalSlotsList)`.

10.1.14 GetPrototypeFromConstructor (*constructor*, *intrinsicDefaultProto*)

The abstract operation `GetPrototypeFromConstructor` takes arguments *constructor* and *intrinsicDefaultProto*. It 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 intrinsic named by *intrinsicDefaultProto* is used for `[[Prototype]]`. It performs the following steps when called:

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. **Assert:** `IsCallable(constructor)` is **true**.
3. Let *proto* be ? `Get(constructor, "prototype")`.
4. If `Type(proto)` is not `Object`, then
 - a. Let *realm* be ? `GetFunctionRealm(constructor)`.
 - b. Set *proto* to *realm*'s intrinsic object named *intrinsicDefaultProto*.
5. Return *proto*.

NOTE

If *constructor* does not supply a `[[Prototype]]` value, the default value that is used is obtained from the [realm](#) of the *constructor* function rather than from the [running execution context](#).

10.1.15 RequireInternalSlot (*O*, *internalSlot*)

The abstract operation `RequireInternalSlot` takes arguments *O* and *internalSlot*. It throws an exception unless *O* is an `Object` and has the given internal slot. It performs the following steps when called:

1. If `Type(O)` is not `Object`, throw a **TypeError** exception.
2. If *O* does not have an *internalSlot* internal slot, throw a **TypeError** exception.

10.2 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 the same internal methods as other ordinary objects. The code of an ECMAScript **function object** may be either **strict mode code** (11.2.2) or **non-strict code**. An ECMAScript **function object** whose code is **strict mode code** is called a *strict function*. One whose code is not **strict mode code** is called a *non-strict function*.

In addition to `[[Extensible]]` and `[[Prototype]]`, ECMAScript function objects also have the internal slots listed in Table 29.

Table 29: Internal Slots of ECMAScript Function Objects

Internal Slot	Type	Description
<code>[[Environment]]</code>	Environment Record	The Environment Record that the function was closed over. Used as the outer environment when evaluating the code of the function.
<code>[[FormalParameters]]</code>	Parse Node	The root parse node of the source text that defines the function's formal parameter list.
<code>[[ECMAScriptCode]]</code>	Parse Node	The root parse node of the source text that defines the function's body.
<code>[[ConstructorKind]]</code>	base derived	Whether or not the function is a derived class constructor .
<code>[[Realm]]</code>	Realm Record	The realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.
<code>[[ScriptOrModule]]</code>	Script Record or Module Record	The script or module in which the function was created.
<code>[[ThisMode]]</code>	lexical strict global	Defines how this references are interpreted within the formal parameters and code body of the function. lexical means that this refers to the this value of a lexically enclosing function. strict means that the this value is used exactly as provided by an invocation of the function. global means that a this value of undefined or null is interpreted as a reference to the global object , and any other this value is first passed to <code>ToObject</code> .
<code>[[Strict]]</code>	Boolean	true if this is a strict function , false if this is a non-strict function .
<code>[[HomeObject]]</code>	Object	If the function uses super , this is the object whose <code>[[GetPrototypeOf]]</code> provides the object where super property lookups begin.
<code>[[SourceText]]</code>	sequence of Unicode code points	The source text that defines the function.
<code>[[IsClassConstructor]]</code>	Boolean	Indicates whether the function is a class constructor . (If true , invoking the function's <code>[[Call]]</code> will immediately throw a TypeError exception.)

All ECMAScript function objects have the `[[Call]]` internal method defined here. ECMAScript functions that are also constructors in addition have the `[[Construct]]` internal method.

10.2.1 `[[Call]]` (*thisArgument*, *argumentsList*)

The `[[Call]]` internal method of an ECMAScript function object *F* takes arguments *thisArgument* (an ECMAScript language value) and *argumentsList* (a List of ECMAScript language values). It performs the following steps when called:

1. **Assert:** *F* is an ECMAScript function object.
2. Let *callerContext* be the running execution context.
3. Let *calleeContext* be `PrepareForOrdinaryCall(F, undefined)`.
4. **Assert:** *calleeContext* is now the running execution context.
5. If *F*.`[[IsClassConstructor]]` is **true**, then
 - a. Let *error* be a newly created **TypeError** object.
 - b. NOTE: *error* is created in *calleeContext* with *F*'s associated **Realm Record**.
 - c. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
 - d. Return `ThrowCompletion(error)`.
6. Perform `OrdinaryCallBindThis(F, calleeContext, thisArgument)`.
7. Let *result* be `OrdinaryCallEvaluateBody(F, argumentsList)`.
8. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
9. If *result*.`[[Type]]` is **return**, return `NormalCompletion(result. [[Value]])`.
10. `ReturnIfAbrupt(result)`.
11. Return `NormalCompletion(undefined)`.

NOTE When *calleeContext* is removed from the execution context stack in step 8 it must not be destroyed if it is suspended and retained for later resumption by an accessible generator object.

10.2.1.1 `PrepareForOrdinaryCall` (*F*, *newTarget*)

The abstract operation `PrepareForOrdinaryCall` takes arguments *F* (a function object) and *newTarget* (an ECMAScript language value). It performs the following steps when called:

1. **Assert:** `Type(newTarget)` is **Undefined** or **Object**.
2. Let *callerContext* be the running execution context.
3. Let *calleeContext* be a new ECMAScript code execution context.
4. Set the **Function** of *calleeContext* to *F*.
5. Let *calleeRealm* be *F*.`[[Realm]]`.
6. Set the **Realm** of *calleeContext* to *calleeRealm*.
7. Set the **ScriptOrModule** of *calleeContext* to *F*.`[[ScriptOrModule]]`.
8. Let *localEnv* be `NewFunctionEnvironment(F, newTarget)`.
9. Set the **LexicalEnvironment** of *calleeContext* to *localEnv*.
10. Set the **VariableEnvironment** of *calleeContext* to *localEnv*.
11. If *callerContext* is not already suspended, suspend *callerContext*.
12. Push *calleeContext* onto the execution context stack; *calleeContext* is now the running execution context.
13. NOTE: Any exception objects produced after this point are associated with *calleeRealm*.

14. Return *calleeContext*.

10.2.1.2 OrdinaryCallBindThis (*F*, *calleeContext*, *thisArgument*)

The abstract operation OrdinaryCallBindThis takes arguments *F* (a [function object](#)), *calleeContext* (an [execution context](#)), and *thisArgument* (an [ECMAScript language value](#)). It performs the following steps when called:

1. Let *thisMode* be *F*.[[ThisMode]].
2. If *thisMode* is [lexical](#), return [NormalCompletion\(undefined\)](#).
3. Let *calleeRealm* be *F*.[[Realm]].
4. Let *localEnv* be the [LexicalEnvironment](#) of *calleeContext*.
5. If *thisMode* is [strict](#), let *thisValue* be *thisArgument*.
6. Else,
 - a. If *thisArgument* is [undefined](#) or [null](#), then
 - i. Let *globalEnv* be *calleeRealm*.[[GlobalEnv]].
 - ii. [Assert](#): *globalEnv* is a [global Environment Record](#).
 - iii. Let *thisValue* be *globalEnv*.[[GlobalThisValue]].
 - b. Else,
 - i. Let *thisValue* be ! [ToObject](#)(*thisArgument*).
 - ii. NOTE: [ToObject](#) produces wrapper objects using *calleeRealm*.
7. [Assert](#): *localEnv* is a [function Environment Record](#).
8. [Assert](#): The next step never returns an [abrupt completion](#) because *localEnv*.[[ThisBindingStatus]] is not initialized.
9. Return *localEnv*.BindThisValue(*thisValue*).

10.2.1.3 Runtime Semantics: EvaluateBody

With parameters *functionObject* and *argumentsList* (a [List](#)).

FunctionBody : *FunctionStatementList*

1. Return ? [EvaluateFunctionBody](#) of *FunctionBody* with arguments *functionObject* and *argumentsList*.

ConciseBody : *ExpressionBody*

1. Return ? [EvaluateConciseBody](#) of *ConciseBody* with arguments *functionObject* and *argumentsList*.

GeneratorBody : *FunctionBody*

1. Return ? [EvaluateGeneratorBody](#) of *GeneratorBody* with arguments *functionObject* and *argumentsList*.

AsyncGeneratorBody : *FunctionBody*

1. Return ? [EvaluateAsyncGeneratorBody](#) of *AsyncGeneratorBody* with arguments *functionObject* and *argumentsList*.

AsyncFunctionBody : *FunctionBody*

1. Return ? [EvaluateAsyncFunctionBody](#) of *AsyncFunctionBody* with arguments *functionObject* and *argumentsList*.

AsyncConciseBody : *ExpressionBody*

1. Return ? **EvaluateAsyncConciseBody** of *AsyncConciseBody* with arguments *functionObject* and *argumentsList*.

10.2.1.4 OrdinaryCallEvaluateBody (*F*, *argumentsList*)

The abstract operation **OrdinaryCallEvaluateBody** takes arguments *F* (a **function object**) and *argumentsList* (a **List**). It performs the following steps when called:

1. Return the result of **EvaluateBody** of the parsed code that is *F*.[[ECMAScriptCode]] passing *F* and *argumentsList* as the arguments.

10.2.2 [[Construct]] (*argumentsList*, *newTarget*)

The [[Construct]] internal method of an ECMAScript **function object** *F* takes arguments *argumentsList* (a **List** of ECMAScript language values) and *newTarget* (a **constructor**). It performs the following steps when called:

1. **Assert**: *F* is an ECMAScript **function object**.
2. **Assert**: **Type**(*newTarget*) is **Object**.
3. Let *callerContext* be the **running execution context**.
4. Let *kind* be *F*.[[ConstructorKind]].
5. If *kind* is **base**, then
 - a. Let *thisArgument* be ? **OrdinaryCreateFromConstructor**(*newTarget*, "%**Object.prototype**%").
6. Let *calleeContext* be **PrepareForOrdinaryCall**(*F*, *newTarget*).
7. **Assert**: *calleeContext* is now the **running execution context**.
8. If *kind* is **base**, perform **OrdinaryCallBindThis**(*F*, *calleeContext*, *thisArgument*).
9. Let *constructorEnv* be the **LexicalEnvironment** of *calleeContext*.
10. Let *result* be **OrdinaryCallEvaluateBody**(*F*, *argumentsList*).
11. Remove *calleeContext* from the **execution context stack** and restore *callerContext* as the **running execution context**.
12. If *result*.[[Type]] is **return**, then
 - a. If **Type**(*result*.[[Value]]) is **Object**, return **NormalCompletion**(*result*.[[Value]]).
 - b. If *kind* is **base**, return **NormalCompletion**(*thisArgument*).
 - c. If *result*.[[Value]] is not **undefined**, throw a **TypeError** exception.
13. Else, **ReturnIfAbrupt**(*result*).
14. Return ? *constructorEnv*.**GetThisBinding**().

10.2.3 OrdinaryFunctionCreate (*functionPrototype*, *sourceText*, *ParameterList*, *Body*, *thisMode*, *Scope*)

The abstract operation **OrdinaryFunctionCreate** takes arguments *functionPrototype* (an **Object**), *sourceText* (a sequence of Unicode code points), *ParameterList* (a **Parse Node**), *Body* (a **Parse Node**), *thisMode* (either **lexical-this** or **non-lexical-this**), and *Scope* (an **Environment Record**). *sourceText* is the source text of the syntactic definition of the function to be created. It performs the following steps when called:

1. **Assert**: **Type**(*functionPrototype*) is **Object**.
2. Let *internalSlotsList* be the internal slots listed in [Table 29](#).
3. Let *F* be ! **OrdinaryObjectCreate**(*functionPrototype*, *internalSlotsList*).
4. Set *F*.[[Call]] to the definition specified in [10.2.1](#).
5. Set *F*.[[SourceText]] to *sourceText*.

6. Set F .[[FormalParameters]] to *ParameterList*.
7. Set F .[[ECMAScriptCode]] to *Body*.
8. If the source text matching *Body* is **strict mode code**, let *Strict* be **true**; else let *Strict* be **false**.
9. Set F .[[Strict]] to *Strict*.
10. If *thisMode* is **lexical-this**, set F .[[ThisMode]] to **lexical**.
11. Else if *Strict* is **true**, set F .[[ThisMode]] to **strict**.
12. Else, set F .[[ThisMode]] to **global**.
13. Set F .[[IsClassConstructor]] to **false**.
14. Set F .[[Environment]] to *Scope*.
15. Set F .[[ScriptOrModule]] to *GetActiveScriptOrModule*() .
16. Set F .[[Realm]] to the **current Realm Record**.
17. Set F .[[HomeObject]] to **undefined**.
18. Let *len* be the *ExpectedArgumentCount* of *ParameterList*.
19. Perform ! *SetFunctionLength*(F , *len*).
20. Return F .

10.2.4 AddRestrictedFunctionProperties (F , *realm*)

The abstract operation *AddRestrictedFunctionProperties* takes arguments F (a **function object**) and *realm* (a **Realm Record**). It performs the following steps when called:

1. **Assert**: *realm*.[[Intrinsics]].[[%ThrowTypeError%]] exists and has been initialized.
2. Let *thrower* be *realm*.[[Intrinsics]].[[%ThrowTypeError%]].
3. Perform ! *DefinePropertyOrThrow*(F , "**caller**", PropertyDescriptor { [[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
4. Return ! *DefinePropertyOrThrow*(F , "**arguments**", PropertyDescriptor { [[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **true** }).

10.2.4.1 %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** }.

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

10.2.5 MakeConstructor (F [, *writablePrototype* [, *prototype*]])

The abstract operation *MakeConstructor* takes argument F (a **function object**) and optional arguments *writablePrototype* (a Boolean) and *prototype* (an Object). It converts F into a **constructor**. It performs the following steps when called:

1. **Assert:** F is an ECMAScript [function object](#) or a built-in [function object](#).
2. If F is an ECMAScript [function object](#), then
 - a. **Assert:** $\text{IsConstructor}(F)$ is **false**.
 - b. **Assert:** F is an extensible object that does not have a **"prototype"** own property.
 - c. Set $F.[[\text{Construct}]]$ to the definition specified in [10.2.2](#).
3. Set $F.[[\text{ConstructorKind}]]$ to **base**.
4. If writablePrototype is not present, set writablePrototype to **true**.
5. If prototype is not present, then
 - a. Set prototype to $! \text{OrdinaryObjectCreate}(\% \text{Object}.\text{prototype}\%)$.
 - b. Perform $! \text{DefinePropertyOrThrow}(\text{prototype}, \text{"constructor"}, \text{PropertyDescriptor} \{ [[\text{Value}]]: F, [[\text{Writable}]]: \text{writablePrototype}, [[\text{Enumerable}]]: \text{false}, [[\text{Configurable}]]: \text{true} \})$.
6. Perform $! \text{DefinePropertyOrThrow}(F, \text{"prototype"}, \text{PropertyDescriptor} \{ [[\text{Value}]]: \text{prototype}, [[\text{Writable}]]: \text{writablePrototype}, [[\text{Enumerable}]]: \text{false}, [[\text{Configurable}]]: \text{false} \})$.
7. Return $\text{NormalCompletion}(\text{undefined})$.

10.2.6 MakeClassConstructor (F)

The abstract operation `MakeClassConstructor` takes argument F . It performs the following steps when called:

1. **Assert:** F is an ECMAScript [function object](#).
2. **Assert:** $F.[[\text{IsClassConstructor}]]$ is **false**.
3. Set $F.[[\text{IsClassConstructor}]]$ to **true**.
4. Return $\text{NormalCompletion}(\text{undefined})$.

10.2.7 MakeMethod ($F, \text{homeObject}$)

The abstract operation `MakeMethod` takes arguments F and homeObject . It configures F as a method. It performs the following steps when called:

1. **Assert:** F is an ECMAScript [function object](#).
2. **Assert:** $\text{Type}(\text{homeObject})$ is `Object`.
3. Set $F.[[\text{HomeObject}]]$ to homeObject .
4. Return $\text{NormalCompletion}(\text{undefined})$.

10.2.8 SetFunctionName (F, name [, prefix])

The abstract operation `SetFunctionName` takes arguments F (a [function object](#)) and name (a property key) and optional argument prefix (a String). It adds a **"name"** property to F . It performs the following steps when called:

1. **Assert:** F is an extensible object that does not have a **"name"** own property.
2. **Assert:** $\text{Type}(\text{name})$ is either `Symbol` or `String`.
3. **Assert:** If prefix is present, then $\text{Type}(\text{prefix})$ is `String`.
4. If $\text{Type}(\text{name})$ is `Symbol`, then
 - a. Let description be name 's $[[\text{Description}]]$ value.
 - b. If description is **undefined**, set name to the empty String.
 - c. Else, set name to the [string-concatenation](#) of `"["`, description , and `"]"`.
5. If F has an $[[\text{InitialName}]]$ internal slot, then
 - a. Set $F.[[\text{InitialName}]]$ to name .

6. If *prefix* is present, then
 - a. Set *name* to the **string-concatenation** of *prefix*, the code unit 0x0020 (SPACE), and *name*.
 - b. If *F* has an `[[InitialName]]` internal slot, then
 - i. Optionally, set *F*.`[[InitialName]]` to *name*.
7. Return ! **DefinePropertyOrThrow**(*F*, "name", PropertyDescriptor { `[[Value]]`: *name*, `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **true** }).

10.2.9 SetFunctionLength (*F*, *length*)

The abstract operation SetFunctionLength takes arguments *F* (a **function object**) and *length* (a non-negative **integer** or $+\infty$). It adds a "length" property to *F*. It performs the following steps when called:

1. **Assert**: *F* is an extensible object that does not have a "length" own property.
2. Return ! **DefinePropertyOrThrow**(*F*, "length", PropertyDescriptor { `[[Value]]`: $\mathbb{F}(length)$, `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **true** }).

10.2.10 FunctionDeclarationInstantiation (*func*, *argumentsList*)

NOTE 1 When an **execution context** is established for evaluating an ECMAScript function a new **function 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.

The abstract operation FunctionDeclarationInstantiation takes arguments *func* (a **function object**) and *argumentsList*. *func* is the **function object** for which the **execution context** is being established. It performs the following steps when called:

1. Let *calleeContext* be the **running execution context**.
2. Let *code* be *func*.`[[ECMAScriptCode]]`.
3. Let *strict* be *func*.`[[Strict]]`.
4. Let *formals* be *func*.`[[FormalParameters]]`.
5. Let *parameterNames* be the **BoundNames** of *formals*.
6. If *parameterNames* has any duplicate entries, let *hasDuplicates* be **true**. Otherwise, let *hasDuplicates* be **false**.
7. Let *simpleParameterList* be **IsSimpleParameterList** of *formals*.
8. Let *hasParameterExpressions* be **ContainsExpression** of *formals*.
9. Let *varNames* be the **VarDeclaredNames** of *code*.
10. Let *varDeclarations* be the **VarScopedDeclarations** of *code*.
11. Let *lexicalNames* be the **LexicallyDeclaredNames** of *code*.
12. Let *functionNames* be a new empty **List**.
13. Let *functionsToInitialize* be a new empty **List**.
14. For each element *d* of *varDeclarations*, in reverse **List** order, do
 - a. If *d* is neither a *VariableDeclaration* nor a *ForBinding* nor a *BindingIdentifier*, then
 - i. **Assert**: *d* is either a *FunctionDeclaration*, a *GeneratorDeclaration*, an *AsyncFunctionDeclaration*, or an

AsyncGeneratorDeclaration.

- 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 function declarations for the same name, the last declaration is used.
 3. Insert *d* as the first element of *functionsToInitialize*.
15. Let *argumentsObjectNeeded* be **true**.
 16. If *func*.[[ThisMode]] is **lexical**, then
 - a. NOTE: Arrow functions never have an arguments objects.
 - b. Set *argumentsObjectNeeded* to **false**.
 17. Else if **"arguments"** is an element of *parameterNames*, then
 - a. Set *argumentsObjectNeeded* to **false**.
 18. Else if *hasParameterExpressions* is **false**, then
 - a. If **"arguments"** is an element of *functionNames* or if **"arguments"** is an element of *lexicalNames*, then
 - i. Set *argumentsObjectNeeded* to **false**.
 19. If *strict* is **true** or if *hasParameterExpressions* is **false**, then
 - a. NOTE: Only a single **Environment Record** is needed for the parameters and top-level vars.
 - b. Let *env* be the **LexicalEnvironment** of *calleeContext*.
 20. Else,
 - a. NOTE: A separate **Environment Record** is needed to ensure that bindings created by **direct eval** calls in the formal parameter list are outside the environment where parameters are declared.
 - b. Let *calleeEnv* be the **LexicalEnvironment** of *calleeContext*.
 - c. Let *env* be **NewDeclarativeEnvironment**(*calleeEnv*).
 - d. **Assert**: The **VariableEnvironment** of *calleeContext* is *calleeEnv*.
 - e. Set the **LexicalEnvironment** of *calleeContext* to *env*.
 21. For each String *paramName* of *parameterNames*, do
 - a. Let *alreadyDeclared* be *env*.**HasBinding**(*paramName*).
 - 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. Perform ! *env*.**CreateMutableBinding**(*paramName*, **false**).
 - ii. If *hasDuplicates* is **true**, then
 1. Perform ! *env*.**InitializeBinding**(*paramName*, **undefined**).
 22. 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: A 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. If *strict* is **true**, then
 - i. Perform ! *env*.**CreateImmutableBinding**(**"arguments"**, **false**).
 - d. Else,
 - i. Perform ! *env*.**CreateMutableBinding**(**"arguments"**, **false**).
 - e. Call *env*.**InitializeBinding**(**"arguments"**, *ao*).
 - f. Let *parameterBindings* be a **List** whose elements are the elements of *parameterNames*, followed by **"arguments"**.

23. Else,
 - a. Let *parameterBindings* be *parameterNames*.
24. Let *iteratorRecord* be `CreateListIteratorRecord(argumentsList)`.
25. If *hasDuplicates* is **true**, then
 - a. Perform ? `IteratorBindingInitialization` for *formals* with *iteratorRecord* and **undefined** as arguments.
26. Else,
 - a. Perform ? `IteratorBindingInitialization` for *formals* with *iteratorRecord* and *env* as arguments.
27. If *hasParameterExpressions* is **false**, then
 - a. NOTE: Only a single `Environment Record` is needed for the parameters and top-level vars.
 - b. Let *instantiatedVarNames* be a copy of the `List parameterBindings`.
 - c. For each element *n* of *varNames*, do
 - i. If *n* is not an element of *instantiatedVarNames*, then
 1. Append *n* to *instantiatedVarNames*.
 2. Perform ! `env.CreateMutableBinding(n, false)`.
 3. Call `env.InitializeBinding(n, undefined)`.
 - d. Let *varEnv* be *env*.
28. 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 *varEnv* be `NewDeclarativeEnvironment(env)`.
 - c. Set the `VariableEnvironment` of *calleeContext* to *varEnv*.
 - d. Let *instantiatedVarNames* be a new empty `List`.
 - e. For each element *n* of *varNames*, do
 - i. If *n* is not an element of *instantiatedVarNames*, then
 1. Append *n* to *instantiatedVarNames*.
 2. Perform ! `varEnv.CreateMutableBinding(n, false)`.
 3. If *n* is not an element of *parameterBindings* or if *n* is an element of *functionNames*, let *initialValue* be **undefined**.
 4. Else,
 - a. Let *initialValue* be ! `env.GetBindingValue(n, false)`.
 5. Call `varEnv.InitializeBinding(n, initialValue)`.
 6. NOTE: A var with the same name as a formal parameter initially has the same value as the corresponding initialized parameter.
29. NOTE: Annex B.3.3.1 adds additional steps at this point.
30. If *strict* is **false**, then
 - a. Let *lexEnv* be `NewDeclarativeEnvironment(varEnv)`.
 - b. NOTE: Non-strict functions use a separate `Environment Record` for top-level lexical declarations so that a `direct eval` can determine whether any var scoped declarations introduced by the eval code conflict with pre-existing top-level lexically scoped declarations. This is not needed for strict functions because a strict `direct eval` always places all declarations into a new `Environment Record`.
31. Else, let *lexEnv* be *varEnv*.
32. Set the `LexicalEnvironment` of *calleeContext* to *lexEnv*.
33. Let *lexDeclarations* be the `LexicallyScopedDeclarations` of *code*.
34. For each element *d* of *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. Perform ! *lexEnv*.CreateImmutableBinding(*dn*, **true**).
- ii. Else,
 1. Perform ! *lexEnv*.CreateMutableBinding(*dn*, **false**).
35. For each **Parse Node** *f* of *functionsToInitialize*, do
 - a. Let *fn* be the sole element of the **BoundNames** of *f*.
 - b. Let *fo* be **InstantiateFunctionObject** of *f* with argument *lexEnv*.
 - c. Perform ! *varEnv*.SetMutableBinding(*fn*, *fo*, **false**).
36. Return **NormalCompletion**(empty).

NOTE 2 **B.3.3** provides an extension to the above algorithm that is necessary for backwards compatibility with web browser implementations of ECMAScript that predate ECMAScript 2015.

NOTE 3 Parameter *Initializers* may contain **direct eval** expressions. Any top level declarations of such evals are only visible to the eval code (11.2). The creation of the environment for such declarations is described in 8.5.3.

10.3 Built-in Function Objects

The built-in function objects defined in this specification may be implemented as either ECMAScript function objects (10.2) whose behaviour is provided using ECMAScript code or as implementation provided function exotic 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 10.1. All such function exotic objects also have `[[Prototype]]`, `[[Extensible]]`, and `[[Realm]]` internal slots.

Unless otherwise specified every built-in **function object** has the `%Function.prototype%` object 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 function body behaviour for both `[[Call]]` and `[[Construct]]` invocations of the function. However, `[[Construct]]` invocation is not supported by all built-in functions. For each built-in function, when invoked with `[[Call]]`, the `[[Call]]` *thisArgument* provides the **this** value, the `[[Call]]` *argumentsList* provides the named parameters, and the `NewTarget` value is **undefined**. When invoked with `[[Construct]]`, the **this** value is uninitialized, the `[[Construct]]` *argumentsList* provides the named parameters, and the `[[Construct]]` *newTarget* parameter provides the `NewTarget` value. If the built-in function is implemented as an ECMAScript **function object** then this specified behaviour must be implemented by the ECMAScript code that is the body of the function. Built-in functions that are ECMAScript function objects must be strict functions. If a built-in **constructor** has any `[[Call]]` behaviour other than throwing a **TypeError** exception, an ECMAScript implementation of the function must be done in a manner that does not cause the function's `[[IsClassConstructor]]` internal slot to have the value **true**.

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.

Built-in functions have an `[[InitialName]]` internal slot.

If a built-in **function object** is not implemented as an ECMAScript function it must provide `[[Call]]` and `[[Construct]]` internal methods that conform to the following definitions:

10.3.1 `[[Call]]` (*thisArgument*, *argumentsList*)

The `[[Call]]` internal method of a built-in **function object** *F* takes arguments *thisArgument* (an ECMAScript language value) and *argumentsList* (a List of ECMAScript language values). It performs the following steps when called:

1. Let *callerContext* be the **running execution context**.
2. If *callerContext* is not already suspended, suspend *callerContext*.
3. Let *calleeContext* be a new **execution context**.
4. Set the Function of *calleeContext* to *F*.
5. Let *calleeRealm* be *F*.`[[Realm]]`.
6. Set the **Realm** of *calleeContext* to *calleeRealm*.
7. Set the ScriptOrModule of *calleeContext* to **null**.
8. Perform any necessary **implementation-defined** initialization of *calleeContext*.
9. Push *calleeContext* onto the **execution context stack**; *calleeContext* is now the **running execution context**.
10. Let *result* be the **Completion Record** that is the result of evaluating *F* in a manner that conforms to the specification of *F*. *thisArgument* is the **this** value, *argumentsList* provides the named parameters, and the **NewTarget** value is **undefined**.
11. Remove *calleeContext* from the **execution context stack** and restore *callerContext* as the **running execution context**.
12. Return *result*.

NOTE When *calleeContext* is removed from the **execution context stack** it must not be destroyed if it has been suspended and retained by an accessible generator object for later resumption.

10.3.2 `[[Construct]]` (*argumentsList*, *newTarget*)

The `[[Construct]]` internal method of a built-in **function object** *F* takes arguments *argumentsList* (a List of ECMAScript language values) and *newTarget* (a **constructor**). The steps performed are the same as `[[Call]]` (see 10.3.1) except that step 10 is replaced by:

10. Let *result* be the **Completion Record** that is the result of evaluating *F* in a manner that conforms to the specification of *F*. The **this** value is uninitialized, *argumentsList* provides the named parameters, and *newTarget* provides the **NewTarget** value.

10.3.3 `CreateBuiltinFunction` (*steps*, *length*, *name*, *internalSlotsList* [, *realm* [, *prototype* [, *prefix*]]])

The abstract operation `CreateBuiltinFunction` takes arguments *steps*, *length*, *name*, and *internalSlotsList* (a List of names of internal slots) and optional arguments *realm*, *prototype*, and *prefix*. *internalSlotsList* contains the names of additional internal slots that must be defined as part of the object. This operation creates a built-in **function object**. It performs the

following steps when called:

1. **Assert:** *steps* is either a set of algorithm steps or other definition of a function's behaviour provided in this specification.
2. If *realm* is not present or *realm* is empty, set *realm* to the current Realm Record.
3. **Assert:** *realm* is a Realm Record.
4. If *prototype* is not present, set *prototype* to *realm*.[[Intrinsics]].[[%Function.prototype%]].
5. Let *func* be a new built-in function object that when called performs the action described by *steps*. The new function object has internal slots whose names are the elements of *internalSlotsList*, and an [[InitialName]] internal slot.
6. Set *func*.[[Realm]] to *realm*.
7. Set *func*.[[Prototype]] to *prototype*.
8. Set *func*.[[Extensible]] to **true**.
9. Set *func*.[[InitialName]] to **null**.
10. Perform ! **SetFunctionLength**(*func*, *length*).
11. If *prefix* is not present, then
 - a. Perform ! **SetFunctionName**(*func*, *name*).
12. Else,
 - a. Perform ! **SetFunctionName**(*func*, *name*, *prefix*).
13. Return *func*.

Each built-in function defined in this specification is created by calling the CreateBuiltinFunction abstract operation.

10.4 Built-in Exotic Object Internal Methods and Slots

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:

10.4.1 Bound Function Exotic Objects

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

An object is a *bound function exotic object* if its [[Call]] and (if applicable) [[Construct]] internal methods use the following implementations, and its other essential internal methods use the definitions found in 10.1. These methods are installed in **BoundFunctionCreate**.

Bound function exotic objects do not have the internal slots of ECMAScript function objects listed in Table 29. Instead they have the internal slots listed in Table 30, in addition to [[Prototype]] and [[Extensible]].

Table 30: Internal Slots of Bound Function Exotic Objects

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

10.4.1.1 [[Call]] (*thisArgument*, *argumentsList*)

The [[Call]] internal method of a [bound function exotic object](#) *F* takes arguments *thisArgument* (an [ECMAScript language value](#)) and *argumentsList* (a [List](#) of ECMAScript language values). It performs the following steps when called:

1. Let *target* be *F*.[[BoundTargetFunction]].
2. Let *boundThis* be *F*.[[BoundThis]].
3. Let *boundArgs* be *F*.[[BoundArguments]].
4. Let *args* be a [List](#) whose elements are the elements of *boundArgs*, followed by the elements of *argumentsList*.
5. Return ? [Call](#)(*target*, *boundThis*, *args*).

10.4.1.2 [[Construct]] (*argumentsList*, *newTarget*)

The [[Construct]] internal method of a [bound function exotic object](#) *F* takes arguments *argumentsList* (a [List](#) of ECMAScript language values) and *newTarget* (a [constructor](#)). It performs the following steps when called:

1. Let *target* be *F*.[[BoundTargetFunction]].
2. **Assert:** [IsConstructor](#)(*target*) is **true**.
3. Let *boundArgs* be *F*.[[BoundArguments]].
4. Let *args* be a [List](#) whose elements are the elements of *boundArgs*, followed by the elements of *argumentsList*.
5. If [SameValue](#)(*F*, *newTarget*) is **true**, set *newTarget* to *target*.
6. Return ? [Construct](#)(*target*, *args*, *newTarget*).

10.4.1.3 BoundFunctionCreate (*targetFunction*, *boundThis*, *boundArgs*)

The abstract operation BoundFunctionCreate takes arguments *targetFunction*, *boundThis*, and *boundArgs*. It is used to specify the creation of new bound function exotic objects. It performs the following steps when called:

1. **Assert:** [Type](#)(*targetFunction*) is Object.
2. Let *proto* be ? [targetFunction](#).[[GetPrototypeOf]]().
3. Let *internalSlotsList* be the internal slots listed in [Table 30](#), plus [[Prototype]] and [[Extensible]].
4. Let *obj* be ! [MakeBasicObject](#)(*internalSlotsList*).
5. Set *obj*.[[Prototype]] to *proto*.
6. Set *obj*.[[Call]] as described in [10.4.1.1](#).
7. If [IsConstructor](#)(*targetFunction*) is **true**, then
 - a. Set *obj*.[[Construct]] as described in [10.4.1.2](#).

8. Set *obj*.[[BoundTargetFunction]] to *targetFunction*.
9. Set *obj*.[[BoundThis]] to *boundThis*.
10. Set *obj*.[[BoundArguments]] to *boundArgs*.
11. Return *obj*.

10.4.2 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 non-configurable **"length"** property whose value is always a non-negative **integral Number** whose **mathematical value** is less than 2^{32} . The value of the **"length"** property is numerically greater than the name of every own property whose name is an **array index**; whenever an own property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever an own property is added whose name is an **array index**, the value of the **"length"** property is changed, if necessary, to be one more than the numeric value of that **array index**; and whenever the value of the **"length"** property is changed, every own property whose name is an **array index** whose value is not smaller than the new length is 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))` equals *P* and `ToUint32(P)` is not the same value as $\mathbb{F}(2^{32} - 1)$.

An object is an *Array exotic object* (or simply, an Array object) if its `[[DefineOwnProperty]]` internal method uses the following implementation, and its other essential internal methods use the definitions found in 10.1. These methods are installed in `ArrayCreate`.

10.4.2.1 `[[DefineOwnProperty]]` (*P*, *Desc*)

The `[[DefineOwnProperty]]` internal method of an **Array exotic object** *A* takes arguments *P* (a property key) and *Desc* (a **Property Descriptor**). It performs the following steps when called:

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:** `! IsDataDescriptor(oldLenDesc)` is **true**.
 - c. **Assert:** `oldLenDesc.[[Configurable]]` is **false**.
 - d. Let *oldLen* be `oldLenDesc.[[Value]]`.
 - e. **Assert:** *oldLen* is a non-negative **integral Number**.
 - f. Let *index* be `! ToUint32(P)`.
 - g. If `index ≥ oldLen` and `oldLenDesc.[[Writable]]` is **false**, return **false**.
 - h. Let *succeeded* be `! OrdinaryDefineOwnProperty(A, P, Desc)`.
 - i. If *succeeded* is **false**, return **false**.
 - j. If `index ≥ oldLen`, then
 - i. Set `oldLenDesc.[[Value]]` to `index + 1F`.
 - ii. Let *succeeded* be `OrdinaryDefineOwnProperty(A, "length", oldLenDesc)`.
 - iii. **Assert:** *succeeded* is **true**.

- k. Return **true**.
4. Return `OrdinaryDefineOwnProperty(A, P, Desc)`.

10.4.2.2 ArrayCreate (*length* [, *proto*])

The abstract operation ArrayCreate takes argument *length* (a non-negative integer) and optional argument *proto*. It is used to specify the creation of new Array exotic objects. It performs the following steps when called:

1. If $length > 2^{32} - 1$, throw a **RangeError** exception.
2. If *proto* is not present, set *proto* to `%Array.prototype%`.
3. Let *A* be ! `MakeBasicObject`(« [[Prototype]], [[Extensible]] »).
4. Set *A*.[[Prototype]] to *proto*.
5. Set *A*.[[DefineOwnProperty]] as specified in 10.4.2.1.
6. Perform ! `OrdinaryDefineOwnProperty`(*A*, "length", PropertyDescriptor { [[Value]]: $\mathbb{F}(length)$, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
7. Return *A*.

10.4.2.3 ArraySpeciesCreate (*originalArray*, *length*)

The abstract operation ArraySpeciesCreate takes arguments *originalArray* and *length* (a non-negative integer). It is used to specify the creation of a new Array object using a `constructor` function that is derived from *originalArray*. It performs the following steps when called:

1. Let *isArray* be ? `isArray`(*originalArray*).
2. If *isArray* is **false**, return ? `ArrayCreate`(*length*).
3. Let *C* be ? `Get`(*originalArray*, "constructor").
4. If `IsConstructor`(*C*) is **true**, then
 - a. Let *thisRealm* be the current Realm Record.
 - b. Let *realmC* be ? `GetFunctionRealm`(*C*).
 - c. If *thisRealm* and *realmC* are not the same Realm Record, then
 - i. If `SameValue`(*C*, *realmC*.[[Intrinsics]].[["%Array%"]]) is **true**, set *C* to **undefined**.
5. If `Type`(*C*) is Object, then
 - a. Set *C* to ? `Get`(*C*, @@species).
 - b. If *C* is **null**, set *C* to **undefined**.
6. If *C* is **undefined**, return ? `ArrayCreate`(*length*).
7. If `IsConstructor`(*C*) is **false**, throw a **TypeError** exception.
8. Return ? `Construct`(*C*, « $\mathbb{F}(length)$ »).

NOTE

If *originalArray* was created using the standard built-in Array `constructor` for a `realm` that is not the `realm` of the `running execution context`, then a new Array is created using the `realm` of the `running execution context`. This maintains compatibility with Web browsers that have historically had that behaviour for the **Array.prototype** methods that now are defined using `ArraySpeciesCreate`.

10.4.2.4 ArraySetLength (*A*, *Desc*)

The abstract operation ArraySetLength takes arguments *A* (an Array object) and *Desc* (a Property Descriptor). It performs the following steps when called:

1. If *Desc*.[[Value]] 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. Let *numberLen* be ? `ToNumber(Desc.[[Value]])`.
5. If *newLen* is not the same value as *numberLen*, throw a **RangeError** exception.
6. Set *newLenDesc*.[[Value]] to *newLen*.
7. Let *oldLenDesc* be `OrdinaryGetOwnProperty(A, "length")`.
8. **Assert**: ! `IsDataDescriptor(oldLenDesc)` is **true**.
9. **Assert**: *oldLenDesc*.[[Configurable]] is **false**.
10. Let *oldLen* be *oldLenDesc*.[[Value]].
11. If *newLen* ≥ *oldLen*, then
 - a. Return `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
12. If *oldLenDesc*.[[Writable]] is **false**, return **false**.
13. If *newLenDesc*.[[Writable]] is absent or has the value **true**, let *newWritable* be **true**.
14. Else,
 - a. NOTE: Setting the [[Writable]] attribute to **false** is deferred in case any elements cannot be deleted.
 - b. Let *newWritable* be **false**.
 - c. Set *newLenDesc*.[[Writable]] to **true**.
15. Let *succeeded* be ! `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
16. If *succeeded* is **false**, return **false**.
17. For each own property key *P* of *A* that is an **array index**, whose numeric value is greater than or equal to *newLen*, in descending numeric index order, do
 - a. Let *deleteSucceeded* be ! *A*.[[Delete]](*P*).
 - b. If *deleteSucceeded* is **false**, then
 - i. Set *newLenDesc*.[[Value]] to ! `ToUint32(P) + 1F`.
 - ii. If *newWritable* is **false**, set *newLenDesc*.[[Writable]] to **false**.
 - iii. Perform ! `OrdinaryDefineOwnProperty(A, "length", newLenDesc)`.
 - iv. Return **false**.
18. If *newWritable* is **false**, then
 - a. Let *succeeded* be ! `OrdinaryDefineOwnProperty(A, "length", PropertyDescriptor { [[Writable]]: false })`.
 - b. **Assert**: *succeeded* is **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.

10.4.3 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. String exotic 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.

An object is a *String exotic object* (or simply, a String object) if its `[[GetOwnProperty]]`, `[[DefineOwnProperty]]`, and `[[OwnPropertyKeys]]` internal methods use the following implementations, and its other essential internal methods

use the definitions found in 10.1. These methods are installed in `StringCreate`.

String exotic objects have the same internal slots as ordinary objects. They also have a `[[StringData]]` internal slot.

10.4.3.1 `[[GetOwnProperty]] (P)`

The `[[GetOwnProperty]]` internal method of a `String exotic object S` takes argument *P* (a property key). It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *desc* be `OrdinaryGetOwnProperty(S, P)`.
3. If *desc* is not **undefined**, return *desc*.
4. Return `!StringGetOwnProperty(S, P)`.

10.4.3.2 `[[DefineOwnProperty]] (P, Desc)`

The `[[DefineOwnProperty]]` internal method of a `String exotic object S` takes arguments *P* (a property key) and *Desc* (a `Property Descriptor`). It performs the following steps when called:

1. **Assert:** `IsPropertyKey(P)` is **true**.
2. Let *stringDesc* be `!StringGetOwnProperty(S, P)`.
3. If *stringDesc* is not **undefined**, then
 - a. Let *extensible* be `S.[[Extensible]]`.
 - b. Return `!IsCompatiblePropertyDescriptor(extensible, Desc, stringDesc)`.
4. Return `!OrdinaryDefineOwnProperty(S, P, Desc)`.

10.4.3.3 `[[OwnPropertyKeys]] ()`

The `[[OwnPropertyKeys]]` internal method of a `String exotic object O` takes no arguments. It performs the following steps when called:

1. Let *keys* be a new empty `List`.
2. Let *str* be `O.[[StringData]]`.
3. **Assert:** `Type(str)` is `String`.
4. Let *len* be the length of *str*.
5. For each `integer i` starting with 0 such that $i < len$, in ascending order, do
 - a. Add `!ToString(F(i))` as the last element of *keys*.
6. For each own property key *P* of *O* such that *P* is an `array index` and `!ToIntegerOrInfinity(P) ≥ len`, in ascending numeric index order, do
 - 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 `array index`, in ascending chronological order of property creation, do
 - 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 ascending chronological order of property creation, do
 - a. Add *P* as the last element of *keys*.
9. Return *keys*.

10.4.3.4 `StringCreate (value, prototype)`

The abstract operation `StringCreate` takes arguments *value* (a String) and *prototype*. It is used to specify the creation of new String exotic objects. It performs the following steps when called:

1. Let *S* be ! `MakeBasicObject`(« `[[Prototype]]`, `[[Extensible]]`, `[[StringData]]` »).
2. Set *S*.`[[Prototype]]` to *prototype*.
3. Set *S*.`[[StringData]]` to *value*.
4. Set *S*.`[[GetOwnProperty]]` as specified in 10.4.3.1.
5. Set *S*.`[[DefineOwnProperty]]` as specified in 10.4.3.2.
6. Set *S*.`[[OwnPropertyKeys]]` as specified in 10.4.3.3.
7. Let *length* be the number of code unit elements in *value*.
8. Perform ! `DefinePropertyOrThrow`(*S*, "length", PropertyDescriptor { `[[Value]]`: $\mathbb{F}(\textit{length})$, `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **false** }).
9. Return *S*.

10.4.3.5 `StringGetOwnProperty` (*S*, *P*)

The abstract operation `StringGetOwnProperty` takes arguments *S* and *P*. It performs the following steps when called:

1. **Assert**: *S* is an Object that has a `[[StringData]]` internal slot.
2. **Assert**: `IsPropertyKey`(*P*) is **true**.
3. If `Type`(*P*) is not String, return **undefined**.
4. Let *index* be ! `CanonicalNumericIndexString`(*P*).
5. If *index* is **undefined**, return **undefined**.
6. If `IsIntegralNumber`(*index*) is **false**, return **undefined**.
7. If *index* is $-0_{\mathbb{F}}$, return **undefined**.
8. Let *str* be *S*.`[[StringData]]`.
9. **Assert**: `Type`(*str*) is String.
10. Let *len* be the length of *str*.
11. If $\mathbb{R}(\textit{index}) < 0$ or $\textit{len} \leq \mathbb{R}(\textit{index})$, return **undefined**.
12. Let *resultStr* be the String value of length 1, containing one code unit from *str*, specifically the code unit at index $\mathbb{R}(\textit{index})$.
13. Return the PropertyDescriptor { `[[Value]]`: *resultStr*, `[[Writable]]`: **false**, `[[Enumerable]]`: **true**, `[[Configurable]]`: **false** }.

10.4.4 Arguments Exotic Objects

Most ECMAScript functions make an arguments object available to their code. Depending upon the characteristics of the function definition, its arguments 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.

An object is an *arguments exotic object* if its internal methods use the following implementations, with the ones not specified here using those found in 10.1. These methods are installed in `CreateMappedArgumentsObject`.

NOTE 1

While `CreateUnmappedArgumentsObject` is grouped into this clause, it creates an **ordinary object**, not an **arguments exotic object**.

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` (20.1.3.6) to identify them as such.

NOTE 2 The `integer`-indexed data properties of an `arguments exotic 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`. If the arguments object is an `ordinary object`, the values of its 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 3 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 4 Ordinary arguments objects define a non-configurable `accessor property` named `"callee"` which throws a `TypeError` exception on access. The `"callee"` property has a more specific meaning for arguments exotic objects, which are created only for some class of non-strict functions. The definition of this property in the ordinary variant exists to ensure that it is not defined in any other manner by conforming ECMAScript implementations.

NOTE 5 ECMAScript implementations of arguments exotic objects have historically contained an `accessor property` named `"caller"`. Prior to ECMAScript 2017, this specification included the definition of a throwing `"caller"` property on ordinary arguments objects. Since implementations do not contain this extension any longer, ECMAScript 2017 dropped the requirement for a throwing `"caller"` accessor.

10.4.4.1 `[[GetOwnProperty]] (P)`

The `[[GetOwnProperty]]` internal method of an `arguments exotic object` *args* takes argument *P* (a property key). It performs the following steps when called:

1. Let *desc* be `OrdinaryGetOwnProperty(args, P)`.
2. If *desc* is `undefined`, return *desc*.
3. Let *map* be `args.[[ParameterMap]]`.
4. Let *isMapped* be `! HasOwnProperty(map, P)`.
5. If *isMapped* is `true`, then
 - a. Set `desc.[[Value]]` to `Get(map, P)`.
6. Return *desc*.

10.4.4.2 `[[DefineOwnProperty]] (P, Desc)`

The `[[DefineOwnProperty]]` internal method of an [arguments exotic object](#) *args* takes arguments *P* (a property key) and *Desc* (a [Property Descriptor](#)). It performs the following steps when called:

1. Let *map* be *args*.[[ParameterMap]].
2. Let *isMapped* be `HasOwnProperty(map, P)`.
3. Let *newArgDesc* be *Desc*.
4. If *isMapped* is **true** and `IsDataDescriptor(Desc)` is **true**, then
 - a. If *Desc*.[[Value]] is not present and *Desc*.[[Writable]] is present and its value is **false**, then
 - i. Set *newArgDesc* to a copy of *Desc*.
 - ii. Set *newArgDesc*.[[Value]] to `Get(map, P)`.
5. Let *allowed* be `? OrdinaryDefineOwnProperty(args, P, newArgDesc)`.
6. If *allowed* is **false**, return **false**.
7. If *isMapped* is **true**, then
 - a. If `IsAccessorDescriptor(Desc)` is **true**, then
 - i. Call *map*.[[Delete]](*P*).
 - b. Else,
 - i. If *Desc*.[[Value]] is present, then
 1. Let *setStatus* be `Set(map, P, Desc.[[Value]], false)`.
 2. **Assert**: *setStatus* 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 *map*.[[Delete]](*P*).
8. Return **true**.

10.4.4.3 `[[Get]]` (*P*, *Receiver*)

The `[[Get]]` internal method of an [arguments exotic object](#) *args* takes arguments *P* (a property key) and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. Let *map* be *args*.[[ParameterMap]].
2. Let *isMapped* be `! HasOwnProperty(map, P)`.
3. If *isMapped* is **false**, then
 - a. Return `? OrdinaryGet(args, P, Receiver)`.
4. Else,
 - a. **Assert**: *map* contains a formal parameter mapping for *P*.
 - b. Return `Get(map, P)`.

10.4.4.4 `[[Set]]` (*P*, *V*, *Receiver*)

The `[[Set]]` internal method of an [arguments exotic object](#) *args* takes arguments *P* (a property key), *V* (an [ECMAScript language value](#)), and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. If `SameValue(args, Receiver)` is **false**, then
 - a. Let *isMapped* be **false**.
2. Else,
 - a. Let *map* be *args*.[[ParameterMap]].
 - b. Let *isMapped* be `! HasOwnProperty(map, P)`.
3. If *isMapped* is **true**, then
 - a. Let *setStatus* be `Set(map, P, V, false)`.

- b. **Assert:** *setStatus* is **true** because formal parameters mapped by argument objects are always writable.
4. Return ? **OrdinarySet**(*args*, *P*, *V*, *Receiver*).

10.4.4.5 **[[Delete]] (P)**

The **[[Delete]]** internal method of an **arguments exotic object** *args* takes argument *P* (a property key). It performs the following steps when called:

1. Let *map* be *args*.**[[ParameterMap]]**.
2. Let *isMapped* be ! **HasOwnProperty**(*map*, *P*).
3. Let *result* be ? **OrdinaryDelete**(*args*, *P*).
4. If *result* is **true** and *isMapped* is **true**, then
 - a. Call *map*.**[[Delete]]**(*P*).
5. Return *result*.

10.4.4.6 **CreateUnmappedArgumentsObject (argumentsList)**

The abstract operation **CreateUnmappedArgumentsObject** takes argument *argumentsList*. It performs the following steps when called:

1. Let *len* be the number of elements in *argumentsList*.
2. Let *obj* be ! **OrdinaryObjectCreate**(%Object.prototype%, « **[[ParameterMap]]** »).
3. Set *obj*.**[[ParameterMap]]** to **undefined**.
4. Perform **DefinePropertyOrThrow**(*obj*, "length", PropertyDescriptor { **[[Value]]**: $\mathbb{F}(\textit{len})$, **[[Writable]]**: **true**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **true** }).
5. Let *index* be 0.
6. Repeat, while *index* < *len*,
 - a. Let *val* be *argumentsList*[*index*].
 - b. Perform ! **CreateDataPropertyOrThrow**(*obj*, ! **Tostring**($\mathbb{F}(\textit{index})$), *val*).
 - c. Set *index* to *index* + 1.
7. Perform ! **DefinePropertyOrThrow**(*obj*, @@iterator, PropertyDescriptor { **[[Value]]**: %Array.prototype.values%, **[[Writable]]**: **true**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **true** }).
8. Perform ! **DefinePropertyOrThrow**(*obj*, "callee", PropertyDescriptor { **[[Get]]**: %ThrowTypeError%, **[[Set]]**: %ThrowTypeError%, **[[Enumerable]]**: **false**, **[[Configurable]]**: **false** }).
9. Return *obj*.

10.4.4.7 **CreateMappedArgumentsObject (func, formals, argumentsList, env)**

The abstract operation **CreateMappedArgumentsObject** takes arguments *func* (an Object), *formals* (a **Parse Node**), *argumentsList* (a **List**), and *env* (an **Environment Record**). It performs the following steps when called:

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 ! **MakeBasicObject**(« **[[Prototype]]**, **[[Extensible]]**, **[[ParameterMap]]** »).
4. Set *obj*.**[[GetOwnProperty]]** as specified in 10.4.4.1.
5. Set *obj*.**[[DefineOwnProperty]]** as specified in 10.4.4.2.
6. Set *obj*.**[[Get]]** as specified in 10.4.4.3.
7. Set *obj*.**[[Set]]** as specified in 10.4.4.4.

8. Set *obj*.[[Delete]] as specified in 10.4.4.5.
9. Set *obj*.[[Prototype]] to %Object.prototype%.
10. Let *map* be ! OrdinaryObjectCreate(**null**).
11. Set *obj*.[[ParameterMap]] to *map*.
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 *argumentsList*[*index*].
 - b. Perform ! CreateDataPropertyOrThrow(*obj*, ! ToString($\mathbb{F}(index)$), *val*).
 - c. Set *index* to *index* + 1.
16. Perform ! DefinePropertyOrThrow(*obj*, "length", PropertyDescriptor { [[Value]]: $\mathbb{F}(len)$, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
17. Let *mappedNames* be a new empty List.
18. Let *index* be *numberOfParameters* - 1.
19. Repeat, while *index* ≥ 0,
 - a. Let *name* be *parameterNames*[*index*].
 - b. If *name* is not an element of *mappedNames*, then
 - i. Add *name* as an element of the list *mappedNames*.
 - ii. If *index* < *len*, then
 1. Let *g* be MakeArgGetter(*name*, *env*).
 2. Let *p* be MakeArgSetter(*name*, *env*).
 3. Perform *map*.[[DefineOwnProperty]](! ToString($\mathbb{F}(index)$), PropertyDescriptor { [[Set]]: *p*, [[Get]]: *g*, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
 - c. Set *index* to *index* - 1.
20. Perform ! DefinePropertyOrThrow(*obj*, @@iterator, PropertyDescriptor { [[Value]]: %Array.prototype.values%, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
21. Perform ! DefinePropertyOrThrow(*obj*, "callee", PropertyDescriptor { [[Value]]: *func*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }).
22. Return *obj*.

10.4.4.7.1 MakeArgGetter (*name*, *env*)

The abstract operation MakeArgGetter takes arguments *name* (a String) and *env* (an Environment Record). It creates a built-in function object that when executed returns the value bound for *name* in *env*. It performs the following steps when called:

1. Let *steps* be the steps of an ArgGetter function as specified below.
2. Let *length* be the number of non-optional parameters of an ArgGetter function as specified below.
3. Let *getter* be ! CreateBuiltinFunction(*steps*, *length*, "", « [[Name]], [[Env]] »).
4. Set *getter*.[[Name]] to *name*.
5. Set *getter*.[[Env]] to *env*.
6. Return *getter*.

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

1. Let *f* be the active function object.
2. Let *name* be *f*.[[Name]].

3. Let *env* be *f*.[[Env]].
4. Return *env*.GetBindingValue(*name*, **false**).

NOTE ArgGetter functions are never directly accessible to ECMAScript code.

10.4.4.7.2 MakeArgSetter (*name*, *env*)

The abstract operation MakeArgSetter takes arguments *name* (a String) and *env* (an [Environment Record](#)). It creates a built-in [function object](#) that when executed sets the value bound for *name* in *env*. It performs the following steps when called:

1. Let *steps* be the steps of an ArgSetter function as specified below.
2. Let *length* be the number of non-optional parameters of an ArgSetter function as specified below.
3. Let *setter* be ! [CreateBuiltinFunction](#)(*steps*, *length*, "", « [[Name]], [[Env]] »).
4. Set *setter*.[[Name]] to *name*.
5. Set *setter*.[[Env]] to *env*.
6. Return *setter*.

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

1. Let *f* be the [active function object](#).
2. Let *name* be *f*.[[Name]].
3. Let *env* be *f*.[[Env]].
4. Return *env*.SetMutableBinding(*name*, *value*, **false**).

NOTE ArgSetter functions are never directly accessible to ECMAScript code.

10.4.5 Integer-Indexed Exotic Objects

An [Integer-Indexed exotic 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 and additionally [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], [[ContentType]], and [[TypedArrayName]] internal slots.

An object is an *Integer-Indexed exotic object* if its [[GetOwnProperty]], [[HasProperty]], [[DefineOwnProperty]], [[Get]], [[Set]], [[Delete]], and [[OwnPropertyKeys]] internal methods use the definitions in this section, and its other essential internal methods use the definitions found in [10.1](#). These methods are installed by [IntegerIndexedObjectCreate](#).

10.4.5.1 [[GetOwnProperty]] (*P*)

The [[GetOwnProperty]] internal method of an [Integer-Indexed exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. **Assert:** IsPropertyKey(*P*) is **true**.
2. **Assert:** *O* is an [Integer-Indexed exotic object](#).
3. If Type(*P*) is String, then
 - a. Let *numericIndex* be ! [CanonicalNumericIndexString](#)(*P*).
 - b. If *numericIndex* is not **undefined**, then

- i. Let *value* be ! `IntegerIndexedElementGet`(*O*, *numericIndex*).
 - ii. If *value* is **undefined**, return **undefined**.
 - iii. Return the PropertyDescriptor { `[[Value]]`: *value*, `[[Writable]]`: **true**, `[[Enumerable]]`: **true**, `[[Configurable]]`: **true** }.
4. Return `OrdinaryGetOwnProperty`(*O*, *P*).

10.4.5.2 `[[HasProperty]]` (*P*)

The `[[HasProperty]]` internal method of an `Integer-Indexed` exotic object *O* takes argument *P* (a property key). It performs the following steps when called:

1. Assert: `IsPropertyKey`(*P*) is **true**.
2. Assert: *O* is an `Integer-Indexed` exotic object.
3. If `Type`(*P*) is String, then
 - a. Let *numericIndex* be ! `CanonicalNumericIndexString`(*P*).
 - b. If *numericIndex* is not **undefined**, return ! `IsValidIntegerIndex`(*O*, *numericIndex*).
4. Return ? `OrdinaryHasProperty`(*O*, *P*).

10.4.5.3 `[[DefineOwnProperty]]` (*P*, *Desc*)

The `[[DefineOwnProperty]]` internal method of an `Integer-Indexed` exotic object *O* takes arguments *P* (a property key) and *Desc* (a `Property Descriptor`). It performs the following steps when called:

1. Assert: `IsPropertyKey`(*P*) is **true**.
2. Assert: *O* is an `Integer-Indexed` exotic object.
3. If `Type`(*P*) is String, then
 - a. Let *numericIndex* be ! `CanonicalNumericIndexString`(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. If ! `IsValidIntegerIndex`(*O*, *numericIndex*) is **false**, return **false**.
 - ii. If *Desc* has a `[[Configurable]]` field and if *Desc*.`[[Configurable]]` is **false**, return **false**.
 - iii. If *Desc* has an `[[Enumerable]]` field and if *Desc*.`[[Enumerable]]` is **false**, return **false**.
 - iv. If ! `IsAccessorDescriptor`(*Desc*) is **true**, return **false**.
 - v. If *Desc* has a `[[Writable]]` field and if *Desc*.`[[Writable]]` is **false**, return **false**.
 - vi. If *Desc* has a `[[Value]]` field, perform ? `IntegerIndexedElementSet`(*O*, *numericIndex*, *Desc*.`[[Value]]`).
 - vii. Return **true**.
4. Return ! `OrdinaryDefineOwnProperty`(*O*, *P*, *Desc*).

10.4.5.4 `[[Get]]` (*P*, *Receiver*)

The `[[Get]]` internal method of an `Integer-Indexed` exotic object *O* takes arguments *P* (a property key) and *Receiver* (an ECMAScript language value). It performs the following steps when called:

1. Assert: `IsPropertyKey`(*P*) is **true**.
2. If `Type`(*P*) is String, then
 - a. Let *numericIndex* be ! `CanonicalNumericIndexString`(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. Return ! `IntegerIndexedElementGet`(*O*, *numericIndex*).
3. Return ? `OrdinaryGet`(*O*, *P*, *Receiver*).

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

The **[[Set]]** internal method of an **Integer-Indexed exotic object** *O* takes arguments *P* (a property key), *V* (an ECMAScript language value), and *Receiver* (an ECMAScript language value). It performs the following steps when called:

1. **Assert**: **IsPropertyKey**(*P*) is **true**.
2. If **Type**(*P*) is String, then
 - a. Let *numericIndex* be ! **CanonicalNumericIndexString**(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. Perform ? **IntegerIndexedElementSet**(*O*, *numericIndex*, *V*).
 - ii. Return **true**.
3. Return ? **OrdinarySet**(*O*, *P*, *V*, *Receiver*).

10.4.5.6 **[[Delete]]** (*P*)

The **[[Delete]]** internal method of an **Integer-Indexed exotic object** *O* takes arguments *P* (a property key). It performs the following steps when called:

1. **Assert**: **IsPropertyKey**(*P*) is **true**.
2. **Assert**: *O* is an **Integer-Indexed exotic object**.
3. If **Type**(*P*) is String, then
 - a. Let *numericIndex* be ! **CanonicalNumericIndexString**(*P*).
 - b. If *numericIndex* is not **undefined**, then
 - i. If ! **IsValidIntegerIndex**(*O*, *numericIndex*) is **false**, return **true**; else return **false**.
4. Return ? **OrdinaryDelete**(*O*, *P*).

10.4.5.7 **[[OwnPropertyKeys]]** ()

The **[[OwnPropertyKeys]]** internal method of an **Integer-Indexed exotic object** *O* takes no arguments. It performs the following steps when called:

1. Let *keys* be a new empty **List**.
2. **Assert**: *O* is an **Integer-Indexed exotic object**.
3. If **IsDetachedBuffer**(*O*.[[ViewedArrayBuffer]]) is **false**, then
 - a. For each **integer** *i* starting with 0 such that $i < O.[[ArrayLength]]$, in ascending order, do
 - i. Add ! **ToString**(**F**(*i*)) as the last element of *keys*.
4. For each own property key *P* of *O* such that **Type**(*P*) is String and *P* is not an **integer index**, in ascending chronological order of property creation, do
 - a. Add *P* as the last element of *keys*.
5. For each own property key *P* of *O* such that **Type**(*P*) is Symbol, in ascending chronological order of property creation, do
 - a. Add *P* as the last element of *keys*.
6. Return *keys*.

10.4.5.8 **IntegerIndexedObjectCreate** (*prototype*)

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

1. Let *internalSlotsList* be « [[Prototype]], [[Extensible]], [[ViewedArrayBuffer]], [[TypedArrayName]], [[ContentType]], [[ByteLength]], [[ByteOffset]], [[ArrayLength]] ».
2. Let *A* be ! **MakeBasicObject**(*internalSlotsList*).
3. Set *A*.[[GetOwnProperty]] as specified in 10.4.5.1.
4. Set *A*.[[HasProperty]] as specified in 10.4.5.2.
5. Set *A*.[[DefineOwnProperty]] as specified in 10.4.5.3.
6. Set *A*.[[Get]] as specified in 10.4.5.4.
7. Set *A*.[[Set]] as specified in 10.4.5.5.
8. Set *A*.[[Delete]] as specified in 10.4.5.6.
9. Set *A*.[[OwnPropertyKeys]] as specified in 10.4.5.7.
10. Set *A*.[[Prototype]] to *prototype*.
11. Return *A*.

10.4.5.9 IsValidIntegerIndex (*O*, *index*)

The abstract operation IsValidIntegerIndex takes arguments *O* and *index* (a Number). It performs the following steps when called:

1. **Assert**: *O* is an Integer-Indexed exotic object.
2. If **IsDetachedBuffer**(*O*.[[ViewedArrayBuffer]]) is **true**, return **false**.
3. If ! **IsIntegralNumber**(*index*) is **false**, return **false**.
4. If *index* is $-0_{\mathbb{F}}$, return **false**.
5. If $\mathbb{R}(\textit{index}) < 0$ or $\mathbb{R}(\textit{index}) \geq O.[[ArrayLength]]$, return **false**.
6. Return **true**.

10.4.5.10 IntegerIndexedElementGet (*O*, *index*)

The abstract operation IntegerIndexedElementGet takes arguments *O* and *index* (a Number). It performs the following steps when called:

1. **Assert**: *O* is an Integer-Indexed exotic object.
2. If ! **IsValidIntegerIndex**(*O*, *index*) is **false**, return **undefined**.
3. Let *offset* be *O*.[[ByteOffset]].
4. Let *arrayTypeName* be the String value of *O*.[[TypedArrayName]].
5. Let *elementSize* be the Element Size value specified in Table 60 for *arrayTypeName*.
6. Let *indexedPosition* be $\mathbb{R}(\textit{index}) \times \textit{elementSize} + \textit{offset}$.
7. Let *elementType* be the Element Type value in Table 60 for *arrayTypeName*.
8. Return **GetValueFromBuffer**(*O*.[[ViewedArrayBuffer]], *indexedPosition*, *elementType*, **true**, **Unordered**).

10.4.5.11 IntegerIndexedElementSet (*O*, *index*, *value*)

The abstract operation IntegerIndexedElementSet takes arguments *O*, *index* (a Number), and *value*. It performs the following steps when called:

1. **Assert**: *O* is an Integer-Indexed exotic object.
2. If *O*.[[ContentType]] is **BigInt**, let *numValue* be ? **ToBigInt**(*value*).
3. Otherwise, let *numValue* be ? **ToNumber**(*value*).
4. If ! **IsValidIntegerIndex**(*O*, *index*) is **true**, then
 - a. Let *offset* be *O*.[[ByteOffset]].

- b. Let *arrayTypeName* be the String value of $O.[[\text{TypedArrayName}]]$.
 - c. Let *elementSize* be the Element Size value specified in Table 60 for *arrayTypeName*.
 - d. Let *indexedPosition* be $(\mathbb{R}(\textit{index}) \times \textit{elementSize}) + \textit{offset}$.
 - e. Let *elementType* be the Element Type value in Table 60 for *arrayTypeName*.
 - f. Perform `SetValueInBuffer`($O.[[\text{ViewedArrayBuffer}]]$, *indexedPosition*, *elementType*, *numValue*, **true**, **Unordered**).
5. Return `NormalCompletion(undefined)`.

NOTE This operation always appears to succeed, but it has no effect when attempting to write past the end of a TypedArray or to a TypedArray which is backed by a detached ArrayBuffer.

10.4.6 Module Namespace Exotic Objects

A **module namespace exotic object** is an **exotic object** that exposes the bindings exported from an ECMAScript *Module* (See 16.2.3). There is a one-to-one correspondence between the String-keyed own properties of a **module namespace exotic object** and the binding names exported by the *Module*. The exported bindings include any bindings that are indirectly exported using **export *** export items. Each String-valued own property key is the **StringValue** of the corresponding exported binding name. These are the only String-keyed properties of a **module namespace exotic object**. Each such property has the attributes { `[[Writable]]: true`, `[[Enumerable]]: true`, `[[Configurable]]: false` }. Module namespace exotic objects are not extensible.

An object is a *module namespace exotic object* if its `[[SetPrototypeOf]]`, `[[IsExtensible]]`, `[[PreventExtensions]]`, `[[GetOwnProperty]]`, `[[DefineOwnProperty]]`, `[[HasProperty]]`, `[[Get]]`, `[[Set]]`, `[[Delete]]`, and `[[OwnPropertyKeys]]` internal methods use the definitions in this section, and its other essential internal methods use the definitions found in 10.1. These methods are installed by `ModuleNamespaceCreate`.

Module namespace exotic objects have the internal slots defined in Table 31.

Table 31: Internal Slots of Module Namespace Exotic Objects

Internal Slot	Type	Description
<code>[[Module]]</code>	Module Record	The Module Record whose exports this namespace exposes.
<code>[[Exports]]</code>	List of String	A List whose elements are the String values of the exported names exposed as own properties of this object. The list is ordered as if an Array of those String values had been sorted using <code>%Array.prototype.sort%</code> using undefined as <i>comparefn</i> .
<code>[[Prototype]]</code>	Null	This slot always contains the value null (see 10.4.6.1).

Module namespace exotic objects provide alternative definitions for all of the internal methods except `[[GetPrototypeOf]]`, which behaves as defined in 10.1.1.

10.4.6.1 `[[SetPrototypeOf]]` (*V*)

The `[[SetPrototypeOf]]` internal method of a **module namespace exotic object** *O* takes argument *V* (an Object or **null**).

It performs the following steps when called:

1. Return ? [SetImmutablePrototype](#)(*O*, *V*).

10.4.6.2 [\[\[IsExtensible\]\]](#) ()

The [\[\[IsExtensible\]\]](#) internal method of a [module namespace exotic object](#) takes no arguments. It performs the following steps when called:

1. Return **false**.

10.4.6.3 [\[\[PreventExtensions\]\]](#) ()

The [\[\[PreventExtensions\]\]](#) internal method of a [module namespace exotic object](#) takes no arguments. It performs the following steps when called:

1. Return **true**.

10.4.6.4 [\[\[GetOwnProperty\]\]](#) (*P*)

The [\[\[GetOwnProperty\]\]](#) internal method of a [module namespace exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. If [Type](#)(*P*) is Symbol, return [OrdinaryGetOwnProperty](#)(*O*, *P*).
2. Let *exports* be *O*.[\[\[Exports\]\]](#).
3. If *P* is not an element of *exports*, return **undefined**.
4. Let *value* be ? *O*.[\[\[Get\]\]](#)(*P*, *O*).
5. Return PropertyDescriptor { [\[\[Value\]\]](#): *value*, [\[\[Writable\]\]](#): **true**, [\[\[Enumerable\]\]](#): **true**, [\[\[Configurable\]\]](#): **false** }.

10.4.6.5 [\[\[DefineOwnProperty\]\]](#) (*P*, *Desc*)

The [\[\[DefineOwnProperty\]\]](#) internal method of a [module namespace exotic object](#) *O* takes arguments *P* (a property key) and *Desc* (a [Property Descriptor](#)). It performs the following steps when called:

1. If [Type](#)(*P*) is Symbol, return [OrdinaryDefineOwnProperty](#)(*O*, *P*, *Desc*).
2. Let *current* be ? *O*.[\[\[GetOwnProperty\]\]](#)(*P*).
3. If *current* is **undefined**, return **false**.
4. If *Desc*.[\[\[Configurable\]\]](#) is present and has value **true**, return **false**.
5. If *Desc*.[\[\[Enumerable\]\]](#) is present and has value **false**, return **false**.
6. If ! [IsAccessorDescriptor](#)(*Desc*) is **true**, return **false**.
7. If *Desc*.[\[\[Writable\]\]](#) is present and has value **false**, return **false**.
8. If *Desc*.[\[\[Value\]\]](#) is present, return [SameValue](#)(*Desc*.[\[\[Value\]\]](#), *current*.[\[\[Value\]\]](#)).
9. Return **true**.

10.4.6.6 [\[\[HasProperty\]\]](#) (*P*)

The [\[\[HasProperty\]\]](#) internal method of a [module namespace exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. If [Type](#)(*P*) is Symbol, return [OrdinaryHasProperty](#)(*O*, *P*).

2. Let *exports* be *O*.[[Exports]].
3. If *P* is an element of *exports*, return **true**.
4. Return **false**.

10.4.6.7 [[Get]] (*P*, *Receiver*)

The [[Get]] internal method of a **module namespace exotic object** *O* takes arguments *P* (a property key) and *Receiver* (an **ECMAScript language value**). It performs the following steps when called:

1. **Assert**: **IsPropertyKey**(*P*) is **true**.
2. If **Type**(*P*) is **Symbol**, then
 - a. Return ? **OrdinaryGet**(*O*, *P*, *Receiver*).
3. Let *exports* be *O*.[[Exports]].
4. If *P* is not an element of *exports*, return **undefined**.
5. Let *m* be *O*.[[Module]].
6. Let *binding* be ! *m*.**ResolveExport**(*P*).
7. **Assert**: *binding* is a **ResolvedBinding Record**.
8. Let *targetModule* be *binding*.[[Module]].
9. **Assert**: *targetModule* is not **undefined**.
10. If *binding*.[[BindingName]] is **"*namespace*"**, then
 - a. Return ? **GetModuleNamespace**(*targetModule*).
11. Let *targetEnv* be *targetModule*.[[Environment]].
12. If *targetEnv* is **undefined**, throw a **ReferenceError** exception.
13. Return ? *targetEnv*.**GetBindingValue**(*binding*.[[BindingName]], **true**).

NOTE **ResolveExport** is side-effect free. Each time this operation is called with a specific *exportName*, *resolveSet* pair as arguments it must return the same result. An implementation might choose to pre-compute or cache the **ResolveExport** results for the [[Exports]] of each **module namespace exotic object**.

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

The [[Set]] internal method of a **module namespace exotic object** takes arguments *P* (a property key), *V* (an **ECMAScript language value**), and *Receiver* (an **ECMAScript language value**). It performs the following steps when called:

1. Return **false**.

10.4.6.9 [[Delete]] (*P*)

The [[Delete]] internal method of a **module namespace exotic object** *O* takes argument *P* (a property key). It performs the following steps when called:

1. **Assert**: **IsPropertyKey**(*P*) is **true**.
2. If **Type**(*P*) is **Symbol**, then
 - a. Return ? **OrdinaryDelete**(*O*, *P*).
3. Let *exports* be *O*.[[Exports]].
4. If *P* is an element of *exports*, return **false**.

5. Return **true**.

10.4.6.10 `[[OwnPropertyKeys]]` ()

The `[[OwnPropertyKeys]]` internal method of a [module namespace exotic object](#) *O* takes no arguments. It performs the following steps when called:

1. Let *exports* be a copy of *O*.`[[Exports]]`.
2. Let *symbolKeys* be `! OrdinaryOwnPropertyKeys(O)`.
3. Append all the entries of *symbolKeys* to the end of *exports*.
4. Return *exports*.

10.4.6.11 `ModuleNamespaceCreate` (*module*, *exports*)

The abstract operation `ModuleNamespaceCreate` takes arguments *module* and *exports*. It is used to specify the creation of new module namespace exotic objects. It performs the following steps when called:

1. **Assert:** *module* is a [Module Record](#).
2. **Assert:** *module*.`[[Namespace]]` is **undefined**.
3. **Assert:** *exports* is a [List](#) of String values.
4. Let *internalSlotsList* be the internal slots listed in [Table 31](#).
5. Let *M* be `! MakeBasicObject(internalSlotsList)`.
6. Set *M*'s essential internal methods to the definitions specified in [10.4.6](#).
7. Set *M*.`[[Prototype]]` to **null**.
8. Set *M*.`[[Module]]` to *module*.
9. Let *sortedExports* be a [List](#) whose elements are the elements of *exports* ordered as if an Array of the same values had been sorted using `%Array.prototype.sort%` using **undefined** as *comparefn*.
10. Set *M*.`[[Exports]]` to *sortedExports*.
11. Create own properties of *M* corresponding to the definitions in [28.3](#).
12. Set *module*.`[[Namespace]]` to *M*.
13. Return *M*.

10.4.7 Immutable Prototype Exotic Objects

An [immutable prototype exotic object](#) is an [exotic object](#) that has a `[[Prototype]]` internal slot that will not change once it is initialized.

An object is an *immutable prototype exotic object* if its `[[SetPrototypeOf]]` internal method uses the following implementation. (Its other essential internal methods may use any implementation, depending on the specific [immutable prototype exotic object](#) in question.)

NOTE

Unlike other exotic objects, there is not a dedicated creation abstract operation provided for immutable prototype exotic objects. This is because they are only used by `%Object.prototype%` and by [host](#) environments, and in [host](#) environments, the relevant objects are potentially exotic in other ways and thus need their own dedicated creation operation.

10.4.7.1 `[[SetPrototypeOf]]` (*V*)

The `[[SetPrototypeOf]]` internal method of an [immutable prototype exotic object](#) O takes argument V (an Object or **null**). It performs the following steps when called:

1. Return `? SetImmutablePrototype(O , V)`.

10.4.7.2 SetImmutablePrototype (O , V)

The abstract operation `SetImmutablePrototype` takes arguments O and V . It performs the following steps when called:

1. **Assert**: Either `Type(V)` is Object or `Type(V)` is Null.
2. Let *current* be `? O .[[GetPrototypeOf]]()`.
3. If `SameValue(V , current)` is **true**, return **true**.
4. Return **false**.

10.5 Proxy Object Internal Methods and Internal Slots

A proxy object is an [exotic object](#) whose essential internal methods are partially implemented using ECMAScript code. Every proxy object has an internal slot called `[[ProxyHandler]]`. The value of `[[ProxyHandler]]` is an object, called the proxy's *handler object*, or **null**. Methods (see [Table 32](#)) 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*.

An object is a *Proxy exotic object* if its essential internal methods (including `[[Call]]` and `[[Construct]]`, if applicable) use the definitions in this section. These internal methods are installed in [ProxyCreate](#).

Table 32: Proxy Handler Methods

Internal Method	Handler Method
[[GetPrototypeOf]]	getPrototypeOf
[[SetPrototypeOf]]	setPrototypeOf
[[IsExtensible]]	isExtensible
[[PreventExtensions]]	preventExtensions
[[GetOwnProperty]]	getOwnPropertyDescriptor
[[DefineOwnProperty]]	defineProperty
[[HasProperty]]	has
[[Get]]	get
[[Set]]	set
[[Delete]]	deleteProperty
[[OwnPropertyKeys]]	ownKeys
[[Call]]	apply
[[Construct]]	construct

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 objects permit the implementation of internal methods to be provided by arbitrary ECMAScript code, 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 object 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 and *Desc* is a [Property Descriptor](#) record.

10.5.1 [[GetPrototypeOf]] ()

The [[GetPrototypeOf]] internal method of a [Proxy exotic object](#) *O* takes no arguments. It performs the following steps

when called:

1. Let *handler* be $O.[[\text{ProxyHandler}]]$.
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** $\text{Type}(\textit{handler})$ is Object.
4. Let *target* be $O.[[\text{ProxyTarget}]]$.
5. Let *trap* be ? $\text{GetMethod}(\textit{handler}, \text{"getPrototypeOf"})$.
6. If *trap* is **undefined**, then
 - a. Return ? $\textit{target}.[[\text{GetPrototypeOf}]>()$.
7. Let *handlerProto* be ? $\text{Call}(\textit{trap}, \textit{handler}, \ll \textit{target} \gg)$.
8. If $\text{Type}(\textit{handlerProto})$ is neither Object nor Null, throw a **TypeError** exception.
9. Let *extensibleTarget* be ? $\text{IsExtensible}(\textit{target})$.
10. If *extensibleTarget* is **true**, return *handlerProto*.
11. Let *targetProto* be ? $\textit{target}.[[\text{GetPrototypeOf}]>()$.
12. If $\text{SameValue}(\textit{handlerProto}, \textit{targetProto})$ is **false**, throw a **TypeError** exception.
13. Return *handlerProto*.

NOTE $[[\text{GetPrototypeOf}]]$ for proxy objects enforces the following invariants:

- The result of $[[\text{GetPrototypeOf}]]$ must be either an Object or **null**.
- If the target object is not extensible, $[[\text{GetPrototypeOf}]]$ applied to the proxy object must return the same value as $[[\text{GetPrototypeOf}]]$ applied to the proxy object's target object.

10.5.2 $[[\text{SetPrototypeOf}]]$ (*V*)

The $[[\text{SetPrototypeOf}]]$ internal method of a **Proxy exotic object** *O* takes argument *V* (an Object or **null**). It performs the following steps when called:

1. **Assert:** Either $\text{Type}(V)$ is Object or $\text{Type}(V)$ is Null.
2. Let *handler* be $O.[[\text{ProxyHandler}]]$.
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert:** $\text{Type}(\textit{handler})$ is Object.
5. Let *target* be $O.[[\text{ProxyTarget}]]$.
6. Let *trap* be ? $\text{GetMethod}(\textit{handler}, \text{"setPrototypeOf"})$.
7. If *trap* is **undefined**, then
 - a. Return ? $\textit{target}.[[\text{SetPrototypeOf}])(V)$.
8. Let *booleanTrapResult* be ! $\text{ToBoolean}(\text{? Call}(\textit{trap}, \textit{handler}, \ll \textit{target}, V \gg))$.
9. If *booleanTrapResult* is **false**, return **false**.
10. Let *extensibleTarget* be ? $\text{IsExtensible}(\textit{target})$.
11. If *extensibleTarget* is **true**, return **true**.
12. Let *targetProto* be ? $\textit{target}.[[\text{GetPrototypeOf}]>()$.
13. If $\text{SameValue}(V, \textit{targetProto})$ is **false**, throw a **TypeError** exception.
14. Return **true**.

NOTE `[[SetPrototypeOf]]` for proxy objects enforces the following invariants:

- The result of `[[SetPrototypeOf]]` is a Boolean value.
- If the target object is not extensible, the argument value must be the same as the result of `[[GetPrototypeOf]]` applied to target object.

10.5.3 `[[IsExtensible]]` ()

The `[[IsExtensible]]` internal method of a [Proxy exotic object](#) *O* takes no arguments. It performs the following steps when called:

1. Let *handler* be *O*.`[[ProxyHandler]]`.
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** `Type(handler)` is Object.
4. Let *target* be *O*.`[[ProxyTarget]]`.
5. Let *trap* be ? `GetMethod(handler, "isExtensible")`.
6. If *trap* is **undefined**, then
 - a. Return ? `IsExtensible(target)`.
7. Let *booleanTrapResult* be ! `ToBoolean(? Call(trap, handler, « target »))`.
8. Let *targetResult* be ? `IsExtensible(target)`.
9. If `SameValue(booleanTrapResult, targetResult)` is **false**, throw a **TypeError** exception.
10. Return *booleanTrapResult*.

NOTE `[[IsExtensible]]` for proxy objects enforces the following invariants:

- The result of `[[IsExtensible]]` is a Boolean value.
- `[[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.

10.5.4 `[[PreventExtensions]]` ()

The `[[PreventExtensions]]` internal method of a [Proxy exotic object](#) *O* takes no arguments. It performs the following steps when called:

1. Let *handler* be *O*.`[[ProxyHandler]]`.
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** `Type(handler)` is Object.
4. Let *target* be *O*.`[[ProxyTarget]]`.
5. Let *trap* be ? `GetMethod(handler, "preventExtensions")`.
6. If *trap* is **undefined**, then
 - a. Return ? *target*.`[[PreventExtensions]]`().
7. Let *booleanTrapResult* be ! `ToBoolean(? Call(trap, handler, « target »))`.
8. If *booleanTrapResult* is **true**, then
 - a. Let *extensibleTarget* be ? `IsExtensible(target)`.
 - b. If *extensibleTarget* is **true**, throw a **TypeError** exception.
9. Return *booleanTrapResult*.

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

- The result of [[PreventExtensions]] is a Boolean value.
- [[PreventExtensions]] applied to the proxy object only returns **true** if [[IsExtensible]] applied to the proxy object's target object is **false**.

10.5.5 [[GetOwnProperty]] (*P*)

The [[GetOwnProperty]] internal method of a [Proxy exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. **Assert:** IsPropertyKey(*P*) is **true**.
2. Let *handler* be *O*.[[ProxyHandler]].
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert:** Type(*handler*) is Object.
5. Let *target* be *O*.[[ProxyTarget]].
6. Let *trap* be ? [GetMethod](#)(*handler*, "getOwnPropertyDescriptor").
7. If *trap* is **undefined**, then
 - a. Return ? *target*.[[GetOwnProperty]](*P*).
8. Let *trapResultObj* be ? [Call](#)(*trap*, *handler*, « *target*, *P* »).
9. If Type(*trapResultObj*) is neither Object nor Undefined, throw a **TypeError** exception.
10. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
11. If *trapResultObj* is **undefined**, then
 - a. If *targetDesc* is **undefined**, return **undefined**.
 - b. If *targetDesc*.[[Configurable]] is **false**, throw a **TypeError** exception.
 - c. Let *extensibleTarget* be ? [IsExtensible](#)(*target*).
 - d. If *extensibleTarget* is **false**, throw a **TypeError** exception.
 - e. Return **undefined**.
12. Let *extensibleTarget* be ? [IsExtensible](#)(*target*).
13. Let *resultDesc* be ? [ToPropertyDescriptor](#)(*trapResultObj*).
14. Call [CompletePropertyDescriptor](#)(*resultDesc*).
15. Let *valid* be [IsCompatiblePropertyDescriptor](#)(*extensibleTarget*, *resultDesc*, *targetDesc*).
16. If *valid* is **false**, throw a **TypeError** exception.
17. If *resultDesc*.[[Configurable]] is **false**, then
 - a. If *targetDesc* is **undefined** or *targetDesc*.[[Configurable]] is **true**, then
 - i. Throw a **TypeError** exception.
 - b. If *resultDesc* has a [[Writable]] field and *resultDesc*.[[Writable]] is **false**, then
 - i. If *targetDesc*.[[Writable]] is **true**, throw a **TypeError** exception.
18. 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 the target object is not extensible, unless it does not exist as an own property of the target object.
- A property cannot be reported as existent, if the target object is not extensible, unless it exists as an own property of the target object.
- A property cannot be reported as non-configurable, unless it exists as a non-configurable own property of the target object.
- A property cannot be reported as both non-configurable and non-writable, unless it exists as a non-configurable, non-writable own property of the target object.

10.5.6 [[DefineOwnProperty]] (*P*, *Desc*)

The [[DefineOwnProperty]] internal method of a **Proxy exotic object** *O* takes arguments *P* (a property key) and *Desc* (a **Property Descriptor**). It performs the following steps when called:

1. **Assert**: **IsPropertyKey**(*P*) is **true**.
2. Let *handler* be *O*.[[ProxyHandler]].
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert**: **Type**(*handler*) is Object.
5. Let *target* be *O*.[[ProxyTarget]].
6. Let *trap* be ? **GetMethod**(*handler*, "defineProperty").
7. If *trap* is **undefined**, then
 - a. Return ? *target*.[[DefineOwnProperty]](*P*, *Desc*).
8. Let *descObj* be **FromPropertyDescriptor**(*Desc*).
9. Let *booleanTrapResult* be ! **ToBoolean**(? **Call**(*trap*, *handler*, « *target*, *P*, *descObj* »)).
10. If *booleanTrapResult* is **false**, return **false**.
11. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
12. Let *extensibleTarget* be ? **IsExtensible**(*target*).
13. If *Desc* has a [[Configurable]] field and if *Desc*.[[Configurable]] is **false**, then
 - a. Let *settingConfigFalse* be **true**.
14. Else, let *settingConfigFalse* be **false**.
15. If *targetDesc* is **undefined**, then
 - a. If *extensibleTarget* is **false**, throw a **TypeError** exception.
 - b. If *settingConfigFalse* is **true**, throw a **TypeError** exception.
16. Else,
 - a. If **IsCompatiblePropertyDescriptor**(*extensibleTarget*, *Desc*, *targetDesc*) is **false**, throw a **TypeError** exception.
 - b. If *settingConfigFalse* is **true** and *targetDesc*.[[Configurable]] is **true**, throw a **TypeError** exception.
 - c. If **IsDataDescriptor**(*targetDesc*) is **true**, *targetDesc*.[[Configurable]] is **false**, and *targetDesc*.[[Writable]] is **true**, then
 - i. If *Desc* has a [[Writable]] field and *Desc*.[[Writable]] is **false**, throw a **TypeError** exception.
17. Return **true**.

NOTE

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

- The result of [[DefineOwnProperty]] is a Boolean value.
- A property cannot be added, if the target object is not extensible.
- A property cannot be non-configurable, unless there exists a corresponding non-configurable own property of the target object.
- A non-configurable property cannot be non-writable, unless there exists a corresponding non-configurable, non-writable own property of the target object.
- If a property has a corresponding target object property then applying the [Property Descriptor](#) of the property to the target object using [[DefineOwnProperty]] will not throw an exception.

10.5.7 [[HasProperty]] (*P*)

The [[HasProperty]] internal method of a [Proxy exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. **Assert:** IsPropertyKey(*P*) is **true**.
2. Let *handler* be *O*.[[ProxyHandler]].
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert:** Type(*handler*) is Object.
5. Let *target* be *O*.[[ProxyTarget]].
6. Let *trap* be ? [GetMethod](#)(*handler*, "has").
7. If *trap* is **undefined**, then
 - a. Return ? *target*.[[HasProperty]](*P*).
8. Let *booleanTrapResult* be ! [ToBoolean](#)(? [Call](#)(*trap*, *handler*, « *target*, *P* »)).
9. If *booleanTrapResult* is **false**, then
 - a. Let *targetDesc* be ? *target*.[[GetOwnProperty]](*P*).
 - b. If *targetDesc* is not **undefined**, then
 - i. If *targetDesc*.[[Configurable]] is **false**, throw a **TypeError** exception.
 - ii. Let *extensibleTarget* be ? [IsExtensible](#)(*target*).
 - iii. If *extensibleTarget* is **false**, throw a **TypeError** exception.
10. Return *booleanTrapResult*.

NOTE

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

- The result of [[HasProperty]] is a Boolean value.
- 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.

10.5.8 [[Get]] (*P*, *Receiver*)

The [[Get]] internal method of a [Proxy exotic object](#) *O* takes arguments *P* (a property key) and *Receiver* (an

ECMAScript language value). It performs the following steps when called:

1. **Assert**: `IsPropertyKey(P)` is **true**.
2. Let *handler* be `O.[[ProxyHandler]]`.
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert**: `Type(handler)` is `Object`.
5. Let *target* be `O.[[ProxyTarget]]`.
6. Let *trap* be `? GetMethod(handler, "get")`.
7. If *trap* is **undefined**, then
 - a. Return `? target.[[Get]](P, Receiver)`.
8. Let *trapResult* be `? Call(trap, handler, « target, P, Receiver »)`.
9. Let *targetDesc* be `? target.[[GetOwnProperty]](P)`.
10. If *targetDesc* is not **undefined** and *targetDesc*.`[[Configurable]]` is **false**, then
 - a. If `IsDataDescriptor(targetDesc)` is **true** and *targetDesc*.`[[Writable]]` is **false**, then
 - i. If `SameValue(trapResult, targetDesc.[[Value]])` is **false**, throw a **TypeError** exception.
 - b. If `IsAccessorDescriptor(targetDesc)` is **true** and *targetDesc*.`[[Get]]` is **undefined**, then
 - i. If *trapResult* is not **undefined**, throw a **TypeError** exception.
11. Return *trapResult*.

NOTE

`[[Get]]` for proxy objects enforces the following invariants:

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

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

The `[[Set]]` internal method of a [Proxy exotic object](#) *O* takes arguments *P* (a property key), *V* (an [ECMAScript language value](#)), and *Receiver* (an [ECMAScript language value](#)). It performs the following steps when called:

1. **Assert**: `IsPropertyKey(P)` is **true**.
2. Let *handler* be `O.[[ProxyHandler]]`.
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert**: `Type(handler)` is `Object`.
5. Let *target* be `O.[[ProxyTarget]]`.
6. Let *trap* be `? GetMethod(handler, "set")`.
7. If *trap* is **undefined**, then
 - a. Return `? target.[[Set]](P, V, Receiver)`.
8. Let *booleanTrapResult* be `! ToBoolean(? Call(trap, handler, « target, P, V, Receiver »))`.
9. If *booleanTrapResult* is **false**, return **false**.
10. Let *targetDesc* be `? target.[[GetOwnProperty]](P)`.
11. If *targetDesc* is not **undefined** and *targetDesc*.`[[Configurable]]` is **false**, then
 - a. If `IsDataDescriptor(targetDesc)` is **true** and *targetDesc*.`[[Writable]]` is **false**, then
 - i. If `SameValue(V, targetDesc.[[Value]])` is **false**, throw a **TypeError** exception.

- b. If `IsAccessorDescriptor(targetDesc)` is **true**, then
 - i. If `targetDesc.[[Set]]` is **undefined**, throw a **TypeError** exception.

12. Return **true**.

NOTE

`[[Set]]` for proxy objects enforces the following invariants:

- The result of `[[Set]]` is a Boolean value.
- 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 own [data property](#).
- Cannot set the value of a property if the corresponding target object property is a non-configurable own [accessor property](#) that has **undefined** as its `[[Set]]` attribute.

10.5.10 `[[Delete]]` (*P*)

The `[[Delete]]` internal method of a [Proxy exotic object](#) *O* takes argument *P* (a property key). It performs the following steps when called:

1. **Assert**: `IsPropertyKey(P)` is **true**.
2. Let *handler* be *O*.`[[ProxyHandler]]`.
3. If *handler* is **null**, throw a **TypeError** exception.
4. **Assert**: `Type(handler)` is `Object`.
5. Let *target* be *O*.`[[ProxyTarget]]`.
6. Let *trap* be ? `GetMethod(handler, "deleteProperty")`.
7. If *trap* is **undefined**, then
 - a. Return ? *target*.`[[Delete]]`(*P*).
8. Let *booleanTrapResult* be ! `ToBoolean(? Call(trap, handler, « target, P »))`.
9. If *booleanTrapResult* is **false**, return **false**.
10. Let *targetDesc* be ? *target*.`[[GetOwnProperty]]`(*P*).
11. If *targetDesc* is **undefined**, return **true**.
12. If *targetDesc*.`[[Configurable]]` is **false**, throw a **TypeError** exception.
13. Let *extensibleTarget* be ? `IsExtensible(target)`.
14. If *extensibleTarget* is **false**, throw a **TypeError** exception.
15. Return **true**.

NOTE

`[[Delete]]` for proxy objects enforces the following invariants:

- The result of `[[Delete]]` is a Boolean value.
- A property cannot be reported as deleted, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as deleted, if it exists as an own property of the target object and the target object is non-extensible.

10.5.11 `[[OwnPropertyKeys]]` ()

The `[[OwnPropertyKeys]]` internal method of a [Proxy exotic object](#) *O* takes no arguments. It performs the following

steps when called:

1. Let *handler* be *O*.[[ProxyHandler]].
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert**: *Type(handler)* is Object.
4. Let *target* be *O*.[[ProxyTarget]].
5. Let *trap* be ? *GetMethod(handler, "ownKeys")*.
6. If *trap* is **undefined**, then
 - a. Return ? *target*.[[OwnPropertyKeys]]().
7. Let *trapResultArray* be ? *Call(trap, handler, « target »)*.
8. Let *trapResult* be ? *CreateListFromArrayLike(trapResultArray, « String, Symbol »)*.
9. If *trapResult* contains any duplicate entries, throw a **TypeError** exception.
10. Let *extensibleTarget* be ? *IsExtensible(target)*.
11. Let *targetKeys* be ? *target*.[[OwnPropertyKeys]]().
12. **Assert**: *targetKeys* is a **List** whose elements are only String and Symbol values.
13. **Assert**: *targetKeys* contains no duplicate entries.
14. Let *targetConfigurableKeys* be a new empty **List**.
15. Let *targetNonconfigurableKeys* be a new empty **List**.
16. For each element *key* of *targetKeys*, do
 - a. Let *desc* be ? *target*.[[GetOwnProperty]](*key*).
 - b. If *desc* is not **undefined** and *desc*.[[Configurable]] is **false**, then
 - i. Append *key* as an element of *targetNonconfigurableKeys*.
 - c. Else,
 - i. Append *key* as an element of *targetConfigurableKeys*.
17. If *extensibleTarget* is **true** and *targetNonconfigurableKeys* is empty, then
 - a. Return *trapResult*.
18. Let *uncheckedResultKeys* be a **List** whose elements are the elements of *trapResult*.
19. For each element *key* of *targetNonconfigurableKeys*, do
 - a. If *key* is not an element of *uncheckedResultKeys*, throw a **TypeError** exception.
 - b. Remove *key* from *uncheckedResultKeys*.
20. If *extensibleTarget* is **true**, return *trapResult*.
21. For each element *key* of *targetConfigurableKeys*, do
 - a. If *key* is not an element of *uncheckedResultKeys*, throw a **TypeError** exception.
 - b. Remove *key* from *uncheckedResultKeys*.
22. If *uncheckedResultKeys* is not empty, throw a **TypeError** exception.
23. Return *trapResult*.

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

- The result of [[OwnPropertyKeys]] is a **List**.
- The returned **List** contains no duplicate entries.
- 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.

10.5.12 `[[Call]]` (*thisArgument*, *argumentsList*)

The `[[Call]]` internal method of a [Proxy exotic object](#) *O* takes arguments *thisArgument* (an ECMAScript language value) and *argumentsList* (a [List](#) of ECMAScript language values). It performs the following steps when called:

1. Let *handler* be *O*.[[ProxyHandler]].
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** `Type(handler)` is Object.
4. Let *target* be *O*.[[ProxyTarget]].
5. Let *trap* be ? `GetMethod(handler, "apply")`.
6. If *trap* is **undefined**, then
 - a. Return ? `Call(target, thisArgument, argumentsList)`.
7. Let *argArray* be ! `CreateArrayFromList(argumentsList)`.
8. Return ? `Call(trap, handler, « target, thisArgument, 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.

10.5.13 `[[Construct]]` (*argumentsList*, *newTarget*)

The `[[Construct]]` internal method of a [Proxy exotic object](#) *O* takes arguments *argumentsList* (a [List](#) of ECMAScript language values) and *newTarget* (a [constructor](#)). It performs the following steps when called:

1. Let *handler* be *O*.[[ProxyHandler]].
2. If *handler* is **null**, throw a **TypeError** exception.
3. **Assert:** `Type(handler)` is Object.
4. Let *target* be *O*.[[ProxyTarget]].
5. **Assert:** `IsConstructor(target)` is **true**.
6. Let *trap* be ? `GetMethod(handler, "construct")`.
7. If *trap* is **undefined**, then
 - a. Return ? `Construct(target, argumentsList, newTarget)`.
8. Let *argArray* be ! `CreateArrayFromList(argumentsList)`.
9. Let *newObj* be ? `Call(trap, handler, « target, argArray, newTarget »)`.
10. If `Type(newObj)` is not Object, throw a **TypeError** exception.
11. Return *newObj*.

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

NOTE 2 `[[Construct]]` for proxy objects enforces the following invariants:

- The result of `[[Construct]]` must be an Object.

10.5.14 `ProxyCreate` (*target*, *handler*)

The abstract operation `ProxyCreate` takes arguments *target* and *handler*. It is used to specify the creation of new [Proxy](#)

exotic objects. It performs the following steps when called:

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 `! MakeBasicObject(« [[ProxyHandler]], [[ProxyTarget]] »)`.
4. Set *P*'s essential internal methods, except for `[[Call]]` and `[[Construct]]`, to the definitions specified in 10.5.
5. If `IsCallable(target)` is `true`, then
 - a. Set *P*.`[[Call]]` as specified in 10.5.12.
 - b. If `IsConstructor(target)` is `true`, then
 - i. Set *P*.`[[Construct]]` as specified in 10.5.13.
6. Set *P*.`[[ProxyTarget]]` to *target*.
7. Set *P*.`[[ProxyHandler]]` to *handler*.
8. Return *P*.

11 ECMAScript Language: Source Code

11.1 Source Text

Syntax

SourceCharacter ::
any Unicode code point

ECMAScript code is expressed using Unicode. 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 source text, or behave as though they were performing normalization of source 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 (LF)) 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 (LF) 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 code point that might terminate the string literal.

11.1.1 Static Semantics: UTF16EncodeCodePoint (*cp*)

The abstract operation UTF16EncodeCodePoint takes argument *cp* (a Unicode code point). It performs the following steps when called:

1. **Assert:** $0 \leq cp \leq 0x10FFFF$.
2. If $cp \leq 0xFFFF$, return the String value consisting of the code unit whose value is *cp*.
3. Let *cu1* be the code unit whose value is $\text{floor}((cp - 0x10000) / 0x400) + 0xD800$.
4. Let *cu2* be the code unit whose value is $((cp - 0x10000) \bmod 0x400) + 0xDC00$.
5. Return the **string-concatenation** of *cu1* and *cu2*.

11.1.2 Static Semantics: CodePointsToString (*text*)

The abstract operation CodePointsToString takes argument *text* (a sequence of Unicode code points). It converts *text* into a String value, as described in 6.1.4. It performs the following steps when called:

1. Let *result* be the empty String.
2. For each code point *cp* of *text*, do
 - a. Set *result* to the **string-concatenation** of *result* and `!UTF16EncodeCodePoint(cp)`.
3. Return *result*.

11.1.3 Static Semantics: UTF16SurrogatePairToCodePoint (*lead*, *trail*)

The abstract operation UTF16SurrogatePairToCodePoint takes arguments *lead* (a code unit) and *trail* (a code unit). Two code units that form a UTF-16 **surrogate pair** are converted to a code point. It performs the following steps when called:

1. **Assert:** *lead* is a **leading surrogate** and *trail* is a **trailing surrogate**.
2. Let *cp* be $(lead - 0xD800) \times 0x400 + (trail - 0xDC00) + 0x10000$.

3. Return the code point *cp*.

11.1.4 Static Semantics: CodePointAt (*string*, *position*)

The abstract operation CodePointAt takes arguments *string* (a String) and *position* (a non-negative integer). It interprets *string* as a sequence of UTF-16 encoded code points, as described in 6.1.4, and reads from it a single code point starting with the code unit at index *position*. It performs the following steps when called:

1. Let *size* be the length of *string*.
2. **Assert:** *position* ≥ 0 and *position* < *size*.
3. Let *first* be the code unit at index *position* within *string*.
4. Let *cp* be the code point whose numeric value is that of *first*.
5. If *first* is not a **leading surrogate** or **trailing surrogate**, then
 - a. Return the **Record** { [[CodePoint]]: *cp*, [[CodeUnitCount]]: 1, [[IsUnpairedSurrogate]]: **false** }.
6. If *first* is a **trailing surrogate** or *position* + 1 = *size*, then
 - a. Return the **Record** { [[CodePoint]]: *cp*, [[CodeUnitCount]]: 1, [[IsUnpairedSurrogate]]: **true** }.
7. Let *second* be the code unit at index *position* + 1 within *string*.
8. If *second* is not a **trailing surrogate**, then
 - a. Return the **Record** { [[CodePoint]]: *cp*, [[CodeUnitCount]]: 1, [[IsUnpairedSurrogate]]: **true** }.
9. Set *cp* to ! UTF16SurrogatePairToCodePoint(*first*, *second*).
10. Return the **Record** { [[CodePoint]]: *cp*, [[CodeUnitCount]]: 2, [[IsUnpairedSurrogate]]: **false** }.

11.1.5 Static Semantics: StringToCodePoints (*string*)

The abstract operation StringToCodePoints takes argument *string* (a String). It returns the sequence of Unicode code points that results from interpreting *string* as UTF-16 encoded Unicode text as described in 6.1.4. It performs the following steps when called:

1. Let *codePoints* be a new empty **List**.
2. Let *size* be the length of *string*.
3. Let *position* be 0.
4. Repeat, while *position* < *size*,
 - a. Let *cp* be ! CodePointAt(*string*, *position*).
 - b. Append *cp*.[[CodePoint]] to *codePoints*.
 - c. Set *position* to *position* + *cp*.[[CodeUnitCount]].
5. Return *codePoints*.

11.1.6 Static Semantics: ParseText (*sourceText*, *goalSymbol*)

The abstract operation ParseText takes arguments *sourceText* (a sequence of Unicode code points) and *goalSymbol* (a nonterminal in one of the ECMAScript grammars). It performs the following steps when called:

1. Attempt to parse *sourceText* using *goalSymbol* as the **goal symbol**, and analyse the parse result for any **early error** conditions. Parsing and **early error** detection may be interleaved in an **implementation-defined** manner.
2. If the parse succeeded and no early errors were found, return the **Parse Node** (an instance of *goalSymbol*) at the root of the parse tree resulting from the parse.
3. Otherwise, return a **List** of one or more **SyntaxError** objects representing the parsing errors and/or early errors. If more than one parsing error or **early error** is present, the number and ordering of error objects in the list is

implementation-defined, but at least one must be present.

NOTE 1 Consider a text that has an **early error** at a particular point, and also a syntax error at a later point. An implementation that does a parse pass followed by an early errors pass might report the syntax error and not proceed to the early errors pass. An implementation that interleaves the two activities might report the **early error** and not proceed to find the syntax error. A third implementation might report both errors. All of these behaviours are conformant.

NOTE 2 See also clause 17.

11.2 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 *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.
- *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]]` and `[[FormalParameters]]` internal slots (see 10.2) of an ECMAScript **function object**. The function code of a particular ECMAScript function does not include any source text that is parsed as the function code of a nested *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

In addition, if the source text referred to above is parsed as:

- the *FormalParameters* and *FunctionBody* of a *FunctionDeclaration* or *FunctionExpression*,
- the *FormalParameters* and *GeneratorBody* of a *GeneratorDeclaration* or *GeneratorExpression*,
- the *FormalParameters* and *AsyncFunctionBody* of an *AsyncFunctionDeclaration* or *AsyncFunctionExpression*,
- or
- the *FormalParameters* and *AsyncGeneratorBody* of an *AsyncGeneratorDeclaration* or *AsyncGeneratorExpression*,

then the source text matching the *BindingIdentifier* (if any) of that declaration or expression is also included in the function code of the corresponding function.

- *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 *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

NOTE 1 Function code is generally provided as the bodies of Function Definitions (15.2), Arrow Function Definitions (15.3), Method Definitions (15.4), Generator Function Definitions (15.5), Async Function Definitions (15.8), Async Generator Function Definitions (15.6), and Async Arrow Functions (15.9). Function code is also derived from the arguments to the Function constructor (20.2.1.1), the GeneratorFunction constructor (27.3.1.1), and the AsyncFunction constructor (27.7.1.1).

NOTE 2 The practical effect of including the *BindingIdentifier* in function code is that the Early Errors for **strict mode code** are applied to a *BindingIdentifier* that is the name of a function whose body contains a "use strict" directive, even if the surrounding code is not **strict mode code**.

11.2.1 Directive Prologues and the Use Strict Directive

A *Directive Prologue* is the longest sequence of *ExpressionStatements* occurring as the initial *StatementListItems* or *ModuleItems* of a *FunctionBody*, a *ScriptBody*, or a *ModuleBody* 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 (12.9). A *Directive Prologue* may be an empty sequence.

A *Use Strict Directive* is an *ExpressionStatement* in a *Directive Prologue* whose *StringLiteral* is either of the exact code point 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 *ExpressionStatements* of a *Directive Prologue* are evaluated normally during evaluation of the containing production. Implementations may define implementation specific meanings for *ExpressionStatements* 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.

11.2.2 Strict Mode Code

An ECMAScript syntactic unit may be processed using either unrestricted or strict mode syntax and semantics (4.3.2). Code is interpreted as *strict mode code* in the following situations:

- Global code is strict mode code if it begins with a *Directive Prologue* that contains a *Use Strict Directive*.
- Module code is always strict mode code.
- All parts of a *ClassDeclaration* or a *ClassExpression* are strict mode code.
- Eval code is strict mode code if it begins with a *Directive Prologue* that contains a *Use Strict Directive* or if the call to **eval** is a **direct eval** that is contained in strict mode code.
- Function code is strict mode code if the associated *FunctionDeclaration*, *FunctionExpression*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *MethodDefinition*, *ArrowFunction*, or *AsyncArrowFunction* is contained in strict mode code or if the code that produces the value of the function's `[[ECMAScriptCode]]` internal slot begins with a

[Directive Prologue](#) that contains a [Use Strict Directive](#).

- Function code that is supplied as the arguments to the built-in `Function`, `Generator`, `AsyncFunction`, and `AsyncGenerator` constructors is strict mode code if the last argument is a `String` that when processed is a `FunctionBody` that begins with a [Directive Prologue](#) that contains a [Use Strict Directive](#).

ECMAScript code that is not strict mode code is called *non-strict code*.

11.2.3 Non-ECMAScript Functions

An ECMAScript implementation may support the evaluation of function exotic objects whose evaluative behaviour is expressed in some [host-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.

12 ECMAScript Language: Lexical Grammar

The source text of an ECMAScript *Script* or *Module* 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 points 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 *InputElementRegExpOrTemplateTail* goal is used in syntactic grammar contexts where a *RegularExpressionLiteral*, a *TemplateMiddle*, or a *TemplateTail* is permitted. The *InputElementRegExp* [goal symbol](#) is used in all syntactic grammar contexts where a *RegularExpressionLiteral* is permitted but neither a *TemplateMiddle*, nor a *TemplateTail* is permitted. The *InputElementTemplateTail* goal is used in all syntactic grammar contexts where a *TemplateMiddle* or a *TemplateTail* is permitted but a *RegularExpressionLiteral* is not permitted. In all other contexts, *InputElementDiv* is used as the lexical [goal symbol](#).

NOTE

The use of multiple lexical goals ensures that there are no lexical ambiguities that would affect automatic semicolon insertion. For example, 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 [12.9](#)); in examples such as the following:

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

where the first non-whitespace, non-comment code point after a *LineTerminator* is U+002F (SOLIDUS) and the syntactic context allows division or division-assignment, no semicolon is inserted at the *LineTerminator*. That is, the above example is interpreted in the same way as:

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

Syntax

InputElementDiv ::

WhiteSpace
LineTerminator
Comment
CommonToken
DivPunctuator
RightBracePunctuator

InputElementRegExp ::

WhiteSpace
LineTerminator
Comment
CommonToken
RightBracePunctuator
RegularExpressionLiteral

InputElementRegExpOrTemplateTail ::

WhiteSpace
LineTerminator
Comment
CommonToken
RegularExpressionLiteral
TemplateSubstitutionTail

InputElementTemplateTail ::

WhiteSpace
LineTerminator
Comment
CommonToken
DivPunctuator
TemplateSubstitutionTail

12.1 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* after the first character.

U+FEFF (ZERO WIDTH NO-BREAK SPACE) 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. <ZWNBSP> 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 `<ZWNBSpace>` code points are treated as white space characters (see 12.2).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarized in Table 33.

Table 33: Format-Control Code Point Usage

Code Point	Name	Abbreviation	Usage
U+200C	ZERO WIDTH NON-JOINER	<code><ZWNJ></code>	<i>IdentifierPart</i>
U+200D	ZERO WIDTH JOINER	<code><ZWJ></code>	<i>IdentifierPart</i>
U+FEFF	ZERO WIDTH NO-BREAK SPACE	<code><ZWNBSpace></code>	<i>WhiteSpace</i>

12.2 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 34.

Table 34: White Space Code Points

Code Point	Name	Abbreviation
U+0009	CHARACTER TABULATION	<code><TAB></code>
U+000B	LINE TABULATION	<code><VT></code>
U+000C	FORM FEED (FF)	<code><FF></code>
U+0020	SPACE	<code><SP></code>
U+00A0	NO-BREAK SPACE	<code><NBSP></code>
U+FEFF	ZERO WIDTH NO-BREAK SPACE	<code><ZWNBSpace></code>
Other category “Zs”	Any other Unicode “Space_Separator” code point	<code><USP></code>

ECMAScript implementations must recognize as *WhiteSpace* code points listed in the “Space_Separator” (“Zs”) category.

NOTE

Other than for the code points listed in Table 34, ECMAScript *WhiteSpace* intentionally excludes all code points that have the Unicode “White_Space” property but which are not classified in category “Space_Separator” (“Zs”).

Syntax

```
WhiteSpace ::  
  <TAB>  
  <VT>  
  <FF>  
  <SP>  
  <NBSP>  
  <ZWNBSP>  
  <USP>
```

12.3 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 (12.9). A line terminator cannot occur within any token except a *StringLiteral*, *Template*, or *TemplateSubstitutionTail*. `<LF>` and `<CR>` line terminators cannot occur within a *StringLiteral* token except as part of a *LineContinuation*.

A line terminator can occur within a *MultiLineComment* 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 35](#).

Table 35: Line Terminator Code Points

Code Point	Unicode Name	Abbreviation
U+000A	LINE FEED (LF)	<code><LF></code>
U+000D	CARRIAGE RETURN (CR)	<code><CR></code>
U+2028	LINE SEPARATOR	<code><LS></code>
U+2029	PARAGRAPH SEPARATOR	<code><PS></code>

Only the Unicode code points in [Table 35](#) 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 34](#). The sequence `<CR><LF>` is commonly used as a line terminator. It should be considered a single *SourceCharacter* for the purpose of reporting line numbers.

Syntax

```
LineTerminator ::  
  <LF>
```

<CR>
<LS>
<PS>

LineTerminatorSequence ::

<LF>
<CR> [lookahead ≠ <LF>]
<LS>
<PS>
<CR> <LF>

12.4 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 [12.9](#)).

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 ::
    // SingleLineCommentCharsopt

SingleLineCommentChars ::
    SingleLineCommentChar SingleLineCommentCharsopt

SingleLineCommentChar ::
    SourceCharacter but not LineTerminator

```

A number of productions in this section are given alternative definitions in section [B.1.3](#)

12.5 Tokens

Syntax

```

CommonToken ::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral
    Template

```

NOTE The *DioPunctuator*, *RegularExpressionLiteral*, *RightBracePunctuator*, and *TemplateSubstitutionTail* productions derive additional tokens that are not included in the *CommonToken* production.

12.6 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 *Identifier* as an *IdentifierName* that is not a *ReservedWord*. The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The Unicode code points in the specified categories in the latest version 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 code points U+200C (ZERO WIDTH NON-JOINER) and U+200D (ZERO WIDTH JOINER) are permitted anywhere after the first code point 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 *CodePoint* of the *UnicodeEscapeSequence* (see [12.8.4](#)). 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 *IdentifierNames* that are canonically equivalent according to the Unicode standard are *not* equal unless, after replacement of each *UnicodeEscapeSequence*, they are represented by the exact same sequence of code points.

Syntax

IdentifierName ::
 IdentifierStart
 IdentifierName *IdentifierPart*

IdentifierStart ::
 UnicodeIDStart
 \$
 –
 \ *UnicodeEscapeSequence*

IdentifierPart ::
 UnicodeIDContinue
 \$
 \ *UnicodeEscapeSequence*
 <ZWJ>
 <ZWJ>

UnicodeIDStart ::
 any Unicode code point with the Unicode property “ID_Start”

UnicodeIDContinue ::
 any Unicode code point with the Unicode property “ID_Continue”

The definitions of the nonterminal *UnicodeEscapeSequence* is given in [12.8.4](#).

NOTE 2 The nonterminal *IdentifierPart* derives `_` via *UnicodeIDContinue*.

NOTE 3 The sets of code points with Unicode properties “ID_Start” and “ID_Continue” include, respectively, the code points with Unicode properties “Other_ID_Start” and “Other_ID_Continue”.

12.6.1 Identifier Names

12.6.1.1 Static Semantics: Early Errors

IdentifierStart :: \ *UnicodeEscapeSequence*

- It is a Syntax Error if the *SV* of *UnicodeEscapeSequence* is none of "\$", or "_", or ! [UTF16EncodeCodePoint\(cp\)](#) for some Unicode code point *cp* matched by the *UnicodeIDStart* lexical grammar production.

IdentifierPart :: \ *UnicodeEscapeSequence*

- It is a Syntax Error if the *SV* of *UnicodeEscapeSequence* is none of "\$", "_", ! *UTF16EncodeCodePoint*(<ZWNJ>), ! *UTF16EncodeCodePoint*(<ZWJ>), or ! *UTF16EncodeCodePoint*(*cp*) for some Unicode code point *cp* that would be matched by the *UnicodeIDContinue* lexical grammar production.

12.6.2 Keywords and Reserved Words

A *keyword* is a token that matches *IdentifierName*, but also has a syntactic use; that is, it appears literally, in a **fixed width** font, in some syntactic production. The keywords of ECMAScript include **if**, **while**, **async**, **await**, and many others.

A *reserved word* is an *IdentifierName* that cannot be used as an identifier. Many keywords are reserved words, but some are not, and some are reserved only in certain contexts. **if** and **while** are reserved words. **await** is reserved only inside async functions and modules. **async** is not reserved; it can be used as a variable name or statement label without restriction.

This specification uses a combination of grammatical productions and [early error](#) rules to specify which names are valid identifiers and which are reserved words. All tokens in the *ReservedWord* list below, except for **await** and **yield**, are unconditionally reserved. Exceptions for **await** and **yield** are specified in [13.1](#), using parameterized syntactic productions. Lastly, several [early error](#) rules restrict the set of valid identifiers. See [13.1.1](#), [14.3.1.1](#), [14.7.5.1](#), and [15.7.1](#). In summary, there are five categories of identifier names:

- Those that are always allowed as identifiers, and are not keywords, such as **Math**, **window**, **toString**, and **–**;
- Those that are never allowed as identifiers, namely the *ReservedWords* listed below except **await** and **yield**;
- Those that are contextually allowed as identifiers, namely **await** and **yield**;
- Those that are contextually disallowed as identifiers, in *strict mode code*: **let**, **static**, **implements**, **interface**, **package**, **private**, **protected**, and **public**;
- Those that are always allowed as identifiers, but also appear as keywords within certain syntactic productions, at places where *Identifier* is not allowed: **as**, **async**, **from**, **get**, **of**, **set**, and **target**.

The term *conditional keyword*, or *contextual keyword*, is sometimes used to refer to the keywords that fall in the last three categories, and thus can be used as identifiers in some contexts and as keywords in others.

Syntax

ReservedWord :: **one of**

```
await break case catch class const continue debugger default delete do else enum export extends
false finally for function if import in instanceof new null return super switch this throw
true try typeof var void while with yield
```

NOTE 1 Per 5.1.5, keywords in the grammar match literal sequences of specific *SourceCharacter* elements. A code point in a keyword cannot be expressed by a `\ UnicodeEscapeSequence`.

An *IdentifierName* can contain `\ UnicodeEscapeSequences`, but it is not possible to declare a variable named "else" by spelling it `els\u{65}`. The *early error* rules in 13.1.1 rule out identifiers with the same *StringValue* as a reserved word.

NOTE 2 **enum** is not currently used as a keyword in this specification. It is a *future reserved word*, set aside for use as a keyword in future language extensions.

Similarly, **implements**, **interface**, **package**, **private**, **protected**, and **public** are future reserved words in *strict mode code*.

NOTE 3 The names **arguments** and **eval** are not keywords, but they are subject to some restrictions in *strict mode code*. See 13.1.1, 8.5.4, 15.2.1, 15.5.1, 15.6.1, and 15.8.1.

12.7 Punctuators

Syntax

Punctuator ::

OptionalChainingPunctuator

OtherPunctuator

OptionalChainingPunctuator ::

? . [lookahead \notin *DecimalDigit*]

OtherPunctuator :: **one of**

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

DivPunctuator ::

/

/=

RightBracePunctuator ::

}

12.8 Literals

12.8.1 Null Literals

Syntax

NullLiteral ::
 null

12.8.2 Boolean Literals

Syntax

BooleanLiteral ::
 true
 false

12.8.3 Numeric Literals

Syntax

NumericLiteralSeparator ::
 -

NumericLiteral ::
 DecimalLiteral
 DecimalBigIntegerLiteral
 NonDecimalIntegerLiteral [+Sep]
 NonDecimalIntegerLiteral [+Sep] *BigIntLiteralSuffix*

DecimalBigIntegerLiteral ::
 0 *BigIntLiteralSuffix*
 NonZeroDigit *DecimalDigits* [+Sep] opt *BigIntLiteralSuffix*
 NonZeroDigit *NumericLiteralSeparator* *DecimalDigits* [+Sep] *BigIntLiteralSuffix*

NonDecimalIntegerLiteral [Sep] ::
 BinaryIntegerLiteral [?Sep]
 OctalIntegerLiteral [?Sep]
 HexIntegerLiteral [?Sep]

BigIntLiteralSuffix ::
 n

DecimalLiteral ::
 DecimalIntegerLiteral . *DecimalDigits* [+Sep] opt *ExponentPart* [+Sep] opt
 . *DecimalDigits* [+Sep] *ExponentPart* [+Sep] opt
 DecimalIntegerLiteral *ExponentPart* [+Sep] opt

DecimalIntegerLiteral ::
 0
 NonZeroDigit
 NonZeroDigit *NumericLiteralSeparator* opt *DecimalDigits* [+Sep]

*DecimalDigits*_[Sep] ::
 DecimalDigit
 *DecimalDigits*_[?Sep] *DecimalDigit*
 [+Sep] *DecimalDigits*_[+Sep] *NumericLiteralSeparator* *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*_[Sep] ::
 ExponentIndicator *SignedInteger*_[?Sep]

ExponentIndicator :: **one of**
 e E

*SignedInteger*_[Sep] ::
 *DecimalDigits*_[?Sep]
 + *DecimalDigits*_[?Sep]
 - *DecimalDigits*_[?Sep]

*BinaryIntegerLiteral*_[Sep] ::
 0b *BinaryDigits*_[?Sep]
 0B *BinaryDigits*_[?Sep]

*BinaryDigits*_[Sep] ::
 BinaryDigit
 *BinaryDigits*_[?Sep] *BinaryDigit*
 [+Sep] *BinaryDigits*_[+Sep] *NumericLiteralSeparator* *BinaryDigit*

BinaryDigit :: **one of**
 0 1

*OctalIntegerLiteral*_[Sep] ::
 0o *OctalDigits*_[?Sep]
 0O *OctalDigits*_[?Sep]

*OctalDigits*_[Sep] ::
 OctalDigit
 *OctalDigits*_[?Sep] *OctalDigit*
 [+Sep] *OctalDigits*_[+Sep] *NumericLiteralSeparator* *OctalDigit*

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

*HexIntegerLiteral*_[Sep] ::

`0x HexDigits` [?Sep]

`0X HexDigits` [?Sep]

`HexDigits` [Sep] `::`

`HexDigit`

`HexDigits` [?Sep] `HexDigit`

[+Sep] `HexDigits` [+Sep] `NumericLiteralSeparator` `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](#), must not extend, as described in [B.1.1](#), the syntax of `NumericLiteral` to include [LegacyOctalIntegerLiteral](#), nor extend the syntax of `DecimalIntegerLiteral` to include [NonOctalDecimalIntegerLiteral](#).

12.8.3.1 Static Semantics: MV

A numeric literal stands for a value of the `Number` type or the `BigInt` type.

- The MV of `NumericLiteral` `::` `DecimalLiteral` is the MV of `DecimalLiteral`.
- The MV of `NonDecimalIntegerLiteral` `::` `BinaryIntegerLiteral` is the MV of `BinaryIntegerLiteral`.
- The MV of `NonDecimalIntegerLiteral` `::` `OctalIntegerLiteral` is the MV of `OctalIntegerLiteral`.
- The MV of `NonDecimalIntegerLiteral` `::` `HexIntegerLiteral` is the MV of `HexIntegerLiteral`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` . is the MV of `DecimalIntegerLiteral`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` . `DecimalDigits` is the MV of `DecimalIntegerLiteral` plus (the MV of `DecimalDigits` $\times 10^{-n}$), where n is the number of code points in `DecimalDigits`, excluding all occurrences of `NumericLiteralSeparator`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` . `ExponentPart` is the MV of `DecimalIntegerLiteral` $\times 10^e$, where e is the MV of `ExponentPart`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` . `DecimalDigits` `ExponentPart` is (the MV of `DecimalIntegerLiteral` plus (the MV of `DecimalDigits` $\times 10^{-n}$)) $\times 10^e$, where n is the number of code points in `DecimalDigits`, excluding all occurrences of `NumericLiteralSeparator` and e is the MV of `ExponentPart`.
- The MV of `DecimalLiteral` `::` . `DecimalDigits` is the MV of `DecimalDigits` $\times 10^{-n}$, where n is the number of code points in `DecimalDigits`, excluding all occurrences of `NumericLiteralSeparator`.
- The MV of `DecimalLiteral` `::` . `DecimalDigits` `ExponentPart` is the MV of `DecimalDigits` $\times 10^{e-n}$, where n is the number of code points in `DecimalDigits`, excluding all occurrences of `NumericLiteralSeparator`, and e is the MV of `ExponentPart`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` is the MV of `DecimalIntegerLiteral`.
- The MV of `DecimalLiteral` `::` `DecimalIntegerLiteral` `ExponentPart` is the MV of `DecimalIntegerLiteral` $\times 10^e$, where e is the MV of `ExponentPart`.
- The MV of `DecimalIntegerLiteral` `::` **0** is 0.
- The MV of `DecimalIntegerLiteral` `::` `NonZeroDigit` is the MV of `NonZeroDigit`.
- The MV of `DecimalIntegerLiteral` `::` `NonZeroDigit` `NumericLiteralSeparator`_{opt} `DecimalDigits` is (the MV of

$NonZeroDigit \times 10^n$) plus the MV of *DecimalDigits*, where *n* is the number of code points in *DecimalDigits*, excluding all occurrences of *NumericLiteralSeparator*.

- The MV of *DecimalDigits* :: *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* :: *DecimalDigits* *DecimalDigit* is (the MV of *DecimalDigits* × 10) plus the MV of *DecimalDigit*.
- The MV of *DecimalDigits* :: *DecimalDigits* *NumericLiteralSeparator* *DecimalDigit* is (the MV of *DecimalDigits* × 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* :: *ExponentIndicator* *SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* :: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :: + *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :: - *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* :: 0 or of *HexDigit* :: 0 or of *OctalDigit* :: 0 or of *BinaryDigit* :: 0 is 0.
- The MV of *DecimalDigit* :: 1 or of *NonZeroDigit* :: 1 or of *HexDigit* :: 1 or of *OctalDigit* :: 1 or of *BinaryDigit* :: 1 is 1.
- The MV of *DecimalDigit* :: 2 or of *NonZeroDigit* :: 2 or of *HexDigit* :: 2 or of *OctalDigit* :: 2 is 2.
- The MV of *DecimalDigit* :: 3 or of *NonZeroDigit* :: 3 or of *HexDigit* :: 3 or of *OctalDigit* :: 3 is 3.
- The MV of *DecimalDigit* :: 4 or of *NonZeroDigit* :: 4 or of *HexDigit* :: 4 or of *OctalDigit* :: 4 is 4.
- The MV of *DecimalDigit* :: 5 or of *NonZeroDigit* :: 5 or of *HexDigit* :: 5 or of *OctalDigit* :: 5 is 5.
- The MV of *DecimalDigit* :: 6 or of *NonZeroDigit* :: 6 or of *HexDigit* :: 6 or of *OctalDigit* :: 6 is 6.
- The MV of *DecimalDigit* :: 7 or of *NonZeroDigit* :: 7 or of *HexDigit* :: 7 or of *OctalDigit* :: 7 is 7.
- The MV of *DecimalDigit* :: 8 or of *NonZeroDigit* :: 8 or of *HexDigit* :: 8 is 8.
- The MV of *DecimalDigit* :: 9 or of *NonZeroDigit* :: 9 or of *HexDigit* :: 9 is 9.
- The MV of *HexDigit* :: a or of *HexDigit* :: A is 10.
- The MV of *HexDigit* :: b or of *HexDigit* :: B is 11.
- The MV of *HexDigit* :: c or of *HexDigit* :: C is 12.
- The MV of *HexDigit* :: d or of *HexDigit* :: D is 13.
- The MV of *HexDigit* :: e or of *HexDigit* :: E is 14.
- The MV of *HexDigit* :: f or of *HexDigit* :: F is 15.
- The MV of *BinaryIntegerLiteral* :: 0b *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 *BinaryDigits* :: *BinaryDigits* *NumericLiteralSeparator* *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 *OctalDigits* :: *OctalDigits* *NumericLiteralSeparator* *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*.
- The MV of *HexDigits* :: *HexDigits* *NumericLiteralSeparator* *HexDigit* is (the MV of *HexDigits* × 16) plus the MV of *HexDigit*.

12.8.3.2 Static Semantics: NumericValue

NumericLiteral :: *DecimalLiteral*

1. Return the **Number value** that results from rounding the MV of *DecimalLiteral* as described below.

NumericLiteral :: *NonDecimalIntegerLiteral*

1. Return the **Number value** that results from rounding the MV of *NonDecimalIntegerLiteral* as described below.

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_F; otherwise, the rounded value must be the **Number value** for the MV (as specified in 6.1.6.1), 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 non-zero digit to its left and there is a non-zero digit, not in the *ExponentPart*, to its right.

NumericLiteral :: *NonDecimalIntegerLiteral* *BigIntLiteralSuffix*

1. Return the BigInt value that represents the MV of *NonDecimalIntegerLiteral*.

DecimalBigIntIntegerLiteral :: 0 *BigIntLiteralSuffix*

1. Return 0_Z.

DecimalBigIntIntegerLiteral :: *NonZeroDigit* *BigIntLiteralSuffix*

1. Return the BigInt value that represents the MV of *NonZeroDigit*.

DecimalBigIntIntegerLiteral ::

NonZeroDigit *DecimalDigits* *BigIntLiteralSuffix*

NonZeroDigit *NumericLiteralSeparator* *DecimalDigits* *BigIntLiteralSuffix*

1. Let *n* be the number of code points in *DecimalDigits*, excluding all occurrences of *NumericLiteralSeparator*.
2. Let *mv* be (the MV of *NonZeroDigit* × 10) plus the MV of *DecimalDigits*.
3. Return $\mathbb{Z}(mv)$.

12.8.4 String Literals

NOTE 1

A string literal is 0 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, U+005C (REVERSE SOLIDUS), U+000D (CARRIAGE RETURN), and U+000A (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 11.1.1. Code points belonging to the Basic Multilingual Plane are encoded as a single code unit element of the string. All other code points are encoded as two code unit elements of the string.

Syntax

StringLiteral ::

" *DoubleStringCharacters*_{opt} "
' *SingleStringCharacters*_{opt} '

DoubleStringCharacters ::

DoubleStringCharacter *DoubleStringCharacters*_{opt}

SingleStringCharacters ::

SingleStringCharacter *SingleStringCharacters*_{opt}

DoubleStringCharacter ::

SourceCharacter but not one of " or \ or *LineTerminator*
<LS>
<PS>
\ *EscapeSequence*
LineContinuation

SingleStringCharacter ::

SourceCharacter but not one of ' or \ or *LineTerminator*
<LS>
<PS>
\ *EscapeSequence*
LineContinuation

LineContinuation ::

\ *LineTerminatorSequence*

EscapeSequence ::

CharacterEscapeSequence
⌀ [lookahead ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

A conforming implementation, when processing [strict mode code](#), must not extend the syntax of *EscapeSequence* to include [LegacyOctalEscapeSequence](#) or [NonOctalDecimalEscapeSequence](#) as described in [B.1.2](#).

CharacterEscapeSequence ::

SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter :: **one of**

' " \ b f n r t v

NonEscapeCharacter ::

SourceCharacter but not one of *EscapeCharacter* or *LineTerminator*

EscapeCharacter ::

SingleEscapeCharacter

DecimalDigit

x

u

HexEscapeSequence ::

x *HexDigit* *HexDigit*

UnicodeEscapeSequence ::

u *Hex4Digits*

u{ *CodePoint* }

Hex4Digits ::

HexDigit *HexDigit* *HexDigit* *HexDigit*

The definition of the nonterminal *HexDigit* is given in 12.8.3. *SourceCharacter* is defined in 11.1.

NOTE 2 <LF> and <CR> cannot appear in a string literal, except as part of a *LineContinuation* to produce the empty code points sequence. The proper way to include either in the String value of a string literal is to use an escape sequence such as `\n` or `\u000A`.

12.8.4.1 Static Semantics: SV

A string literal stands for a value of the String type. The String value (SV) of the literal is described in terms of String values contributed by the various parts of the string literal. As part of this process, some Unicode code points within the string literal are interpreted as having a [mathematical value](#) (MV), as described below or in 12.8.3.

- The SV of *StringLiteral* :: " " is the empty String.
- The SV of *StringLiteral* :: ' ' is the empty String.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* *DoubleStringCharacters* is the [string-concatenation](#) of the SV of *DoubleStringCharacter* and the SV of *DoubleStringCharacters*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* *SingleStringCharacters* is the [string-concatenation](#) of the SV of *SingleStringCharacter* and the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacter* :: *SourceCharacter* but not one of " or \ or *LineTerminator* is the result of performing [UTF16EncodeCodePoint](#) on the code point value of *SourceCharacter*.
- The SV of *DoubleStringCharacter* :: <LS> is the String value consisting of the code unit 0x2028 (LINE SEPARATOR).
- The SV of *DoubleStringCharacter* :: <PS> is the String value consisting of the code unit 0x2029 (PARAGRAPH SEPARATOR).
- The SV of *DoubleStringCharacter* :: *LineContinuation* is the empty String.
- The SV of *SingleStringCharacter* :: *SourceCharacter* but not one of ' or \ or *LineTerminator* is the result of performing [UTF16EncodeCodePoint](#) on the code point value of *SourceCharacter*.
- The SV of *SingleStringCharacter* :: <LS> is the String value consisting of the code unit 0x2028 (LINE SEPARATOR).
- The SV of *SingleStringCharacter* :: <PS> is the String value consisting of the code unit 0x2029 (PARAGRAPH SEPARATOR).
- The SV of *SingleStringCharacter* :: *LineContinuation* is the empty String.
- The SV of *EscapeSequence* :: **0** is the String value consisting of the code unit 0x0000 (NULL).

- The SV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the String value consisting of the code unit whose value is determined by the *SingleEscapeCharacter* according to [Table 36](#).

Table 36: String Single Character Escape Sequences

Escape Sequence	Code Unit Value	Unicode Character Name	Symbol
<code>\b</code>	<code>0x0008</code>	BACKSPACE	<BS>
<code>\t</code>	<code>0x0009</code>	CHARACTER TABULATION	<HT>
<code>\n</code>	<code>0x000A</code>	LINE FEED (LF)	<LF>
<code>\v</code>	<code>0x000B</code>	LINE TABULATION	<VT>
<code>\f</code>	<code>0x000C</code>	FORM FEED (FF)	<FF>
<code>\r</code>	<code>0x000D</code>	CARRIAGE RETURN (CR)	<CR>
<code>\"</code>	<code>0x0022</code>	QUOTATION MARK	"
<code>\'</code>	<code>0x0027</code>	APOSTROPHE	'
<code>\\</code>	<code>0x005C</code>	REVERSE SOLIDUS	\

- The SV of *NonEscapeCharacter* :: *SourceCharacter* but not one of *EscapeCharacter* or *LineTerminator* is the result of performing [UTF16EncodeCodePoint](#) on the code point value of *SourceCharacter*.
- The SV of *HexEscapeSequence* :: `x HexDigit HexDigit` is the String value consisting of the code unit whose value is the MV of *HexEscapeSequence*.
- The SV of *Hex4Digits* :: `HexDigit HexDigit HexDigit HexDigit` is the String value consisting of the code unit whose value is the MV of *Hex4Digits*.
- The SV of *UnicodeEscapeSequence* :: `u{ CodePoint }` is the result of performing [UTF16EncodeCodePoint](#) on the MV of *CodePoint*.

12.8.4.2 Static Semantics: MV

- The MV of *HexEscapeSequence* :: `x HexDigit HexDigit` is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The MV of *Hex4Digits* :: `HexDigit HexDigit HexDigit HexDigit` is (0x1000 × the MV of the first *HexDigit*) plus (0x100 × the MV of the second *HexDigit*) plus (0x10 × the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

12.8.5 Regular Expression Literals

NOTE 1

A regular expression literal is an input element that is converted to a `RegExp` object (see [22.2](#)) 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` or calling the `RegExp` constructor as a function (see [22.2.3](#)).

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The source text comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are subsequently parsed again using the more stringent ECMAScript Regular Expression grammar (22.2.1).

An implementation may extend the ECMAScript Regular Expression grammar defined in 22.2.1, 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]

NOTE 2 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: `/(?:)/`.

12.8.5.1 Static Semantics: Early Errors

RegularExpressionFlags :: *RegularExpressionFlags* *IdentifierPart*

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

12.8.5.2 Static Semantics: BodyText

RegularExpressionLiteral :: / *RegularExpressionBody* / *RegularExpressionFlags*

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

12.8.5.3 Static Semantics: FlagText

RegularExpressionLiteral :: / *RegularExpressionBody* / *RegularExpressionFlags*

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

12.8.6 Template Literal Lexical Components

Syntax

Template ::
NoSubstitutionTemplate
TemplateHead

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

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

TemplateSubstitutionTail ::
TemplateMiddle
TemplateTail

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

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

TemplateCharacters ::
TemplateCharacter *TemplateCharacters*_{opt}

TemplateCharacter ::

\$ [lookahead ≠ {]

\ *EscapeSequence*

\ *NotEscapeSequence*

LineContinuation

LineTerminatorSequence

SourceCharacter but not one of ` or \ or \$ or *LineTerminator*

NotEscapeSequence ::

0 *DecimalDigit*

DecimalDigit but not 0

x [lookahead ∉ *HexDigit*]

x *HexDigit* [lookahead ∉ *HexDigit*]

u [lookahead ∉ *HexDigit*] [lookahead ≠ {]

u *HexDigit* [lookahead ∉ *HexDigit*]

u *HexDigit HexDigit* [lookahead ∉ *HexDigit*]

u *HexDigit HexDigit HexDigit* [lookahead ∉ *HexDigit*]

u { [lookahead ∉ *HexDigit*]

u { *NotCodePoint* [lookahead ∉ *HexDigit*]

u { *CodePoint* [lookahead ∉ *HexDigit*] [lookahead ≠]}

NotCodePoint ::

*HexDigits*_[~Sep] but only if MV of *HexDigits* > 0x10FFFF

CodePoint ::

*HexDigits*_[~Sep] but only if MV of *HexDigits* ≤ 0x10FFFF

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.

12.8.6.1 Static Semantics: TV and TRV

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 String values (SV, 12.8.4) 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, 12.8.3). 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 String.
- The TV and TRV of *TemplateHead* :: ` \${ is the empty String.
- The TV and TRV of *TemplateMiddle* :: } \${ is the empty String.
- The TV and TRV of *TemplateTail* :: } ` is the empty String.
- The TV of *TemplateCharacters* :: *TemplateCharacter TemplateCharacters* is **undefined** if either the TV of *TemplateCharacter* is **undefined** or the TV of *TemplateCharacters* is **undefined**. Otherwise, it is the **string-**

concatenation of the TV of *TemplateCharacter* and the TV of *TemplateCharacters*.

- The TV of *TemplateCharacter* :: *SourceCharacter* but not one of ` or \ or \$ or *LineTerminator* is the result of performing [UTF16EncodeCodePoint](#) on the code point value of *SourceCharacter*.
- The TV of *TemplateCharacter* :: \$ is the String value consisting of the code unit 0x0024 (DOLLAR SIGN).
- The TV of *TemplateCharacter* :: \ *EscapeSequence* is the [SV](#) of *EscapeSequence*.
- The TV of *TemplateCharacter* :: \ *NotEscapeSequence* is **undefined**.
- The TV of *TemplateCharacter* :: *LineTerminatorSequence* is the TRV of *LineTerminatorSequence*.
- The TV of *LineContinuation* :: \ *LineTerminatorSequence* is the empty String.
- The TRV of *TemplateCharacters* :: *TemplateCharacter* *TemplateCharacters* is the [string-concatenation](#) of the TRV of *TemplateCharacter* and the TRV of *TemplateCharacters*.
- The TRV of *TemplateCharacter* :: *SourceCharacter* but not one of ` or \ or \$ or *LineTerminator* is the result of performing [UTF16EncodeCodePoint](#) on the code point value of *SourceCharacter*.
- The TRV of *TemplateCharacter* :: \$ is the String value consisting of the code unit 0x0024 (DOLLAR SIGN).
- The TRV of *TemplateCharacter* :: \ *EscapeSequence* is the [string-concatenation](#) of the code unit 0x005C (REVERSE SOLIDUS) and the TRV of *EscapeSequence*.
- The TRV of *TemplateCharacter* :: \ *NotEscapeSequence* is the [string-concatenation](#) of the code unit 0x005C (REVERSE SOLIDUS) and the TRV of *NotEscapeSequence*.
- The TRV of *EscapeSequence* :: 0 is the String value consisting of the code unit 0x0030 (DIGIT ZERO).
- The TRV of *NotEscapeSequence* :: 0 *DecimalDigit* is the [string-concatenation](#) of the code unit 0x0030 (DIGIT ZERO) and the TRV of *DecimalDigit*.
- The TRV of *NotEscapeSequence* :: x [lookahead ∉ *HexDigit*] is the String value consisting of the code unit 0x0078 (LATIN SMALL LETTER X).
- The TRV of *NotEscapeSequence* :: x *HexDigit* [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0078 (LATIN SMALL LETTER X) and the TRV of *HexDigit*.
- The TRV of *NotEscapeSequence* :: u [lookahead ∉ *HexDigit*] [lookahead ≠ {] is the String value consisting of the code unit 0x0075 (LATIN SMALL LETTER U).
- The TRV of *NotEscapeSequence* :: u *HexDigit* [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U) and the TRV of *HexDigit*.
- The TRV of *NotEscapeSequence* :: u *HexDigit* *HexDigit* [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U), the TRV of the first *HexDigit*, and the TRV of the second *HexDigit*.
- The TRV of *NotEscapeSequence* :: u *HexDigit* *HexDigit* *HexDigit* [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U), the TRV of the first *HexDigit*, the TRV of the second *HexDigit*, and the TRV of the third *HexDigit*.
- The TRV of *NotEscapeSequence* :: u { [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U) and the code unit 0x007B (LEFT CURLY BRACKET).
- The TRV of *NotEscapeSequence* :: u { *NotCodePoint* [lookahead ∉ *HexDigit*] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U), the code unit 0x007B (LEFT CURLY BRACKET), and the TRV of *NotCodePoint*.
- The TRV of *NotEscapeSequence* :: u { *CodePoint* [lookahead ∉ *HexDigit*] [lookahead ≠ }] is the [string-concatenation](#) of the code unit 0x0075 (LATIN SMALL LETTER U), the code unit 0x007B (LEFT CURLY BRACKET), and the TRV of *CodePoint*.
- The TRV of *DecimalDigit* :: **one of** 0 1 2 3 4 5 6 7 8 9 is the result of performing [UTF16EncodeCodePoint](#) on the single code point matched by this production.
- The TRV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the [SV](#) of *NonEscapeCharacter*.
- The TRV of *SingleEscapeCharacter* :: **one of** ' " \ b f n r t v is the result of performing [UTF16EncodeCodePoint](#) on the single code point matched by this production.
- The TRV of *HexEscapeSequence* :: x *HexDigit* *HexDigit* is the [string-concatenation](#) of the code unit 0x0078

(LATIN SMALL LETTER X), the TRV of the first *HexDigit*, and the TRV of the second *HexDigit*.

- The TRV of *UnicodeEscapeSequence* :: **u** *Hex4Digits* is the **string-concatenation** of the code unit 0x0075 (LATIN SMALL LETTER U) and the TRV of *Hex4Digits*.
- The TRV of *UnicodeEscapeSequence* :: **u**{ *CodePoint* } is the **string-concatenation** of the code unit 0x0075 (LATIN SMALL LETTER U), the code unit 0x007B (LEFT CURLY BRACKET), the TRV of *CodePoint*, and the code unit 0x007D (RIGHT CURLY BRACKET).
- The TRV of *Hex4Digits* :: *HexDigit HexDigit HexDigit HexDigit* is the **string-concatenation** of the TRV of the first *HexDigit*, the TRV of the second *HexDigit*, the TRV of the third *HexDigit*, and the TRV of the fourth *HexDigit*.
- The TRV of *HexDigits* :: *HexDigits HexDigit* is the **string-concatenation** of the TRV of *HexDigits* and the TRV of *HexDigit*.
- The TRV of *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** is the result of performing **UTF16EncodeCodePoint** on the single code point matched by this production.
- The TRV of *LineContinuation* :: **** *LineTerminatorSequence* is the **string-concatenation** of the code unit 0x005C (REVERSE SOLIDUS) and the TRV of *LineTerminatorSequence*.
- The TRV of *LineTerminatorSequence* :: **<LF>** is the String value consisting of the code unit 0x000A (LINE FEED).
- The TRV of *LineTerminatorSequence* :: **<CR>** is the String value consisting of the code unit 0x000A (LINE FEED).
- The TRV of *LineTerminatorSequence* :: **<LS>** is the String value consisting of the code unit 0x2028 (LINE SEPARATOR).
- The TRV of *LineTerminatorSequence* :: **<PS>** is the String value consisting of the code unit 0x2029 (PARAGRAPH SEPARATOR).
- The TRV of *LineTerminatorSequence* :: **<CR> <LF>** is the String value consisting of the code unit 0x000A (LINE FEED).

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.

12.9 Automatic Semicolon Insertion

Most ECMAScript statements and declarations must be terminated with a semicolon. 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.

12.9.1 Rules of Automatic Semicolon Insertion

In the following rules, “token” means the actual recognized lexical token determined using the current lexical **goal symbol** as described in clause 12.

There are three basic rules of semicolon insertion:

1. When, as the source text 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 `}`.
 - The previous token is `)` and the inserted semicolon would then be parsed as the terminating semicolon of a do-while statement (14.7.2).
2. When, as the source text 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 instance of the goal nonterminal, then a semicolon is automatically inserted at the end of the input stream.
 3. When, as the source text 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 14.7.4).

NOTE

The following are the only restricted productions in the grammar:

```
UpdateExpression[Yield, Await] :  
    LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] ++  
    LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] --  
ContinueStatement[Yield, Await] :  
    continue ;  
    continue [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;  
BreakStatement[Yield, Await] :  
    break ;  
    break [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;  
ReturnStatement[Yield, Await] :  
    return ;  
    return [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;  
ThrowStatement[Yield, Await] :  
    throw [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;  
ArrowFunction[In, Yield, Await] :  
    ArrowParameters[?Yield, ?Await] [no LineTerminator here] => ConciseBody[?In]  
YieldExpression[In, Await] :  
    yield  
    yield [no LineTerminator here] AssignmentExpression[?In, +Yield, ?Await]  
    yield [no LineTerminator here] * AssignmentExpression[?In, +Yield, ?Await]
```

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.
- A *LabelIdentifier* in a **break** or **continue** statement should be on the same line as the **break** or **continue** token.

12.9.2 Examples of Automatic Semicolon Insertion

This section is non-normative.

The source

```
{ 1 2 } 3
```

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

```
{ 1  
2 } 3
```

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

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

which is a valid ECMAScript sentence.

The source

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

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

The source

```
return  
a + b
```

is transformed by automatic semicolon insertion into the following:

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

NOTE 1

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

The source

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

is transformed by automatic semicolon insertion into the following:

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

NOTE 2

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

The source

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

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

The source

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

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

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

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

12.9.3 Interesting Cases of Automatic Semicolon Insertion

This section is non-normative.

ECMAScript programs can be written in a style with very few semicolons by relying on automatic semicolon insertion. As described above, semicolons are not inserted at every newline, and automatic semicolon insertion can depend on multiple tokens across line terminators.

As new syntactic features are added to ECMAScript, additional grammar productions could be added that cause lines relying on automatic semicolon insertion preceding them to change grammar productions when parsed.

For the purposes of this section, a case of automatic semicolon insertion is considered interesting if it is a place where a semicolon may or may not be inserted, depending on the source text which precedes it. The rest of this section describes a number of interesting cases of automatic semicolon insertion in this version of ECMAScript.

12.9.3.1 Interesting Cases of Automatic Semicolon Insertion in Statement Lists

In a *StatementList*, many *StatementListItem*s end in semicolons, which may be omitted using automatic semicolon insertion. As a consequence of the rules above, at the end of a line ending an expression, a semicolon is required if the following line begins with any of the following:

- **An opening parenthesis (C).** Without a semicolon, the two lines together are treated as a *CallExpression*.
- **An opening square bracket (D).** Without a semicolon, the two lines together are treated as property access, rather than an *ArrayLiteral* or *ArrayAssignmentPattern*.
- **A template literal (`).** Without a semicolon, the two lines together are interpreted as a tagged Template (13.3.11), with the previous expression as the *MemberExpression*.
- **Unary + or -.** Without a semicolon, the two lines together are interpreted as a usage of the corresponding binary operator.
- **A RegExp literal.** Without a semicolon, the two lines together may be parsed instead as the *MultiplicativeOperator*, for example if the RegExp has flags.

12.9.3.2 Cases of Automatic Semicolon Insertion and “[no *LineTerminator* here]”

This section is non-normative.

ECMAScript contains grammar productions which include “[no *LineTerminator* here]”. These productions are sometimes a means to have optional operands in the grammar. Introducing a *LineTerminator* in these locations would change the grammar production of a source text by using the grammar production without the optional operand.

The rest of this section describes a number of productions using “[no *LineTerminator* here]” in this version of ECMAScript.

12.9.3.2.1 List of Grammar Productions with Optional Operands and “[no *LineTerminator* here]”

- *UpdateExpression*.
- *ContinueStatement*.
- *BreakStatement*.
- *ReturnStatement*.
- *YieldExpression*.
- Async Function Definitions (15.8) with relation to Function Definitions (15.2)

13 ECMAScript Language: Expressions

13.1 Identifiers

Syntax

IdentifierReference [*Yield*, *Await*] :

Identifier

[~*Yield*] **yield**

[~*Await*] **await**

BindingIdentifier [*Yield*, *Await*] :

Identifier

yield

await

LabelIdentifier [*Yield*, *Await*] :

Identifier

[~*Yield*] **yield**

[~*Await*] **await**

Identifier :

IdentifierName but not *ReservedWord*

NOTE

yield and **await** are permitted as *BindingIdentifier* in the grammar, and prohibited with [static semantics](#) below, to prohibit automatic semicolon insertion in cases such as

```
let  
await 0;
```

13.1.1 Static Semantics: Early Errors

BindingIdentifier : **Identifier**

- It is a Syntax Error if the code matched by this production is contained in [strict mode code](#) and the [StringValue](#) of *Identifier* is "arguments" or "eval".

IdentifierReference : **yield**

BindingIdentifier : **yield**

LabelIdentifier : **yield**

- It is a Syntax Error if the code matched by this production is contained in [strict mode code](#).

IdentifierReference : **await**

BindingIdentifier : **await**

LabelIdentifier : **await**

- It is a Syntax Error if the [goal symbol](#) of the syntactic grammar is *Module*.

BindingIdentifier [[Yield](#), [Await](#)] : **yield**

- It is a Syntax Error if this production has a [[Yield](#)] parameter.

BindingIdentifier [[Yield](#), [Await](#)] : **await**

- It is a Syntax Error if this production has an [[Await](#)] parameter.

IdentifierReference [[Yield](#), [Await](#)] : *Identifier*

BindingIdentifier [[Yield](#), [Await](#)] : *Identifier*

LabelIdentifier [[Yield](#), [Await](#)] : *Identifier*

- It is a Syntax Error if this production has a [[Yield](#)] parameter and [StringValue](#) of *Identifier* is "yield".
- It is a Syntax Error if this production has an [[Await](#)] parameter and [StringValue](#) of *Identifier* is "await".

Identifier : *IdentifierName* but not *ReservedWord*

- It is a Syntax Error if this phrase is contained in [strict mode code](#) and the [StringValue](#) of *IdentifierName* is: "implements", "interface", "let", "package", "private", "protected", "public", "static", or "yield".
- It is a Syntax Error if the [goal symbol](#) of the syntactic grammar is *Module* and the [StringValue](#) of *IdentifierName* is "await".
- It is a Syntax Error if [StringValue](#) of *IdentifierName* is the same String value as the [StringValue](#) of any *ReservedWord* except for **yield** or **await**.

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`.

13.1.2 Static Semantics: StringValue

`IdentifierName` ::

`IdentifierStart`

`IdentifierName` `IdentifierPart`

1. Let `idText` be the source text matched by `IdentifierName`.
2. Let `idTextUnescaped` be the result of replacing any occurrences of `\ UnicodeEscapeSequence` in `idText` with the code point represented by the `UnicodeEscapeSequence`.
3. Return ! `CodePointsToString(idTextUnescaped)`.

`IdentifierReference` : **yield**

`BindingIdentifier` : **yield**

`LabelIdentifier` : **yield**

1. Return "yield".

`IdentifierReference` : **await**

`BindingIdentifier` : **await**

`LabelIdentifier` : **await**

1. Return "await".

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

1. Return the `StringValue` of `IdentifierName`.

13.1.3 Runtime Semantics: Evaluation

`IdentifierReference` : `Identifier`

1. Return ? `ResolveBinding(StringValue of Identifier)`.

`IdentifierReference` : **yield**

1. Return ? `ResolveBinding("yield")`.

`IdentifierReference` : **await**

1. Return ? `ResolveBinding("await")`.

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` 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`.

13.2 Primary Expression

Syntax

*PrimaryExpression*_[Yield, Await] :

- this**
- IdentifierReference*_[?Yield, ?Await]
- Literal*
- ArrayLiteral*_[?Yield, ?Await]
- ObjectLiteral*_[?Yield, ?Await]
- FunctionExpression*
- ClassExpression*_[?Yield, ?Await]
- GeneratorExpression*
- AsyncFunctionExpression*
- AsyncGeneratorExpression*
- RegularExpressionLiteral*
- TemplateLiteral*_[?Yield, ?Await, ~Tagged]
- CoverParenthesizedExpressionAndArrowParameterList*_[?Yield, ?Await]

*CoverParenthesizedExpressionAndArrowParameterList*_[Yield, Await] :

- (*Expression*_[+In, ?Yield, ?Await])
- (*Expression*_[+In, ?Yield, ?Await] ,)
- ()
- (... *BindingIdentifier*_[?Yield, ?Await])
- (... *BindingPattern*_[?Yield, ?Await])
- (*Expression*_[+In, ?Yield, ?Await] , ... *BindingIdentifier*_[?Yield, ?Await])
- (*Expression*_[+In, ?Yield, ?Await] , ... *BindingPattern*_[?Yield, ?Await])

Supplemental Syntax

When processing an instance of the production

*PrimaryExpression*_[Yield, Await] : *CoverParenthesizedExpressionAndArrowParameterList*_[?Yield, ?Await]
the interpretation of *CoverParenthesizedExpressionAndArrowParameterList* is refined using the following grammar:

*ParenthesizedExpression*_[Yield, Await] :

- (*Expression*_[+In, ?Yield, ?Await])

13.2.1 Semantics

13.2.1.1 Static Semantics: CoveredParenthesizedExpression

CoverParenthesizedExpressionAndArrowParameterList : (*Expression*)

1. Return the *ParenthesizedExpression* that is **covered** by *CoverParenthesizedExpressionAndArrowParameterList*.

13.2.2 The **this** Keyword

13.2.2.1 Runtime Semantics: Evaluation

PrimaryExpression : **this**

1. Return ? [ResolveThisBinding\(\)](#).

13.2.3 Identifier Reference

See [13.1](#) for *IdentifierReference*.

13.2.4 Literals

Syntax

Literal :

NullLiteral

BooleanLiteral

NumericLiteral

StringLiteral

13.2.4.1 Runtime Semantics: Evaluation

Literal : *NullLiteral*

1. Return **null**.

Literal : *BooleanLiteral*

1. If *BooleanLiteral* is the token **false**, return **false**.
2. If *BooleanLiteral* is the token **true**, return **true**.

Literal : *NumericLiteral*

1. Return the [NumericValue](#) of *NumericLiteral* as defined in [12.8.3](#).

Literal : *StringLiteral*

1. Return the [SV](#) of *StringLiteral* as defined in [12.8.4.1](#).

13.2.5 Array Initializer

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*_[Yield, Await] :

[*Elision*_{opt}]

[*ElementList*_[?Yield, ?Await]]

[*ElementList*_[?Yield, ?Await] , *Elision*_{opt}]

*ElementList*_[Yield, Await] :

*Elision*_{opt} *AssignmentExpression*_[+In, ?Yield, ?Await]

*Elision*_{opt} *SpreadElement*_[?Yield, ?Await]

*ElementList*_[?Yield, ?Await] , *Elision*_{opt} *AssignmentExpression*_[+In, ?Yield, ?Await]

*ElementList*_[?Yield, ?Await] , *Elision*_{opt} *SpreadElement*_[?Yield, ?Await]

Elision :

,

Elision ,

*SpreadElement*_[Yield, Await] :

... *AssignmentExpression*_[+In, ?Yield, ?Await]

13.2.5.1 Runtime Semantics: ArrayAccumulation

With parameters *array* and *nextIndex*.

Elision : ,

1. Let *len* be *nextIndex* + 1.
2. Perform ? **Set**(*array*, "length", $\mathbb{F}(\textit{len})$, true).
3. NOTE: The above Set throws if *len* exceeds $2^{32}-1$.
4. Return *len*.

Elision : *Elision* ,

1. Return the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex* + 1.

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

1. If *Elision* is present, then
 - a. Set *nextIndex* to the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex*.
 - b. **ReturnIfAbrupt**(*nextIndex*).
2. Let *initResult* be the result of evaluating *AssignmentExpression*.
3. Let *initValue* be ? **GetValue**(*initResult*).
4. Let *created* be ! **CreateDataPropertyOrThrow**(*array*, ! **ToString**($\mathbb{F}(\textit{nextIndex})$), *initValue*).
5. Return *nextIndex* + 1.

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

1. If *Elision* is present, then
 - a. Set *nextIndex* to the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex*.
 - b. **ReturnIfAbrupt**(*nextIndex*).
2. Return the result of performing **ArrayAccumulation** for *SpreadElement* with arguments *array* and *nextIndex*.

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

1. Set *nextIndex* to the result of performing **ArrayAccumulation** for *ElementList* with arguments *array* and *nextIndex*.
2. **ReturnIfAbrupt**(*nextIndex*).
3. If *Elision* is present, then
 - a. Set *nextIndex* to the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex*.
 - b. **ReturnIfAbrupt**(*nextIndex*).
4. Let *initResult* be the result of evaluating *AssignmentExpression*.
5. Let *initValue* be ? **GetValue**(*initResult*).
6. Let *created* be ! **CreateDataPropertyOrThrow**(*array*, ! **ToString**(**F**(*nextIndex*)), *initValue*).
7. Return *nextIndex* + 1.

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

1. Set *nextIndex* to the result of performing **ArrayAccumulation** for *ElementList* with arguments *array* and *nextIndex*.
2. **ReturnIfAbrupt**(*nextIndex*).
3. If *Elision* is present, then
 - a. Set *nextIndex* to the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex*.
 - b. **ReturnIfAbrupt**(*nextIndex*).
4. Return the result of performing **ArrayAccumulation** for *SpreadElement* with arguments *array* and *nextIndex*.

SpreadElement : ... *AssignmentExpression*

1. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
2. Let *spreadObj* be ? **GetValue**(*spreadRef*).
3. Let *iteratorRecord* be ? **GetIterator**(*spreadObj*).
4. Repeat,
 - a. Let *next* be ? **IteratorStep**(*iteratorRecord*).
 - b. If *next* is **false**, return *nextIndex*.
 - c. Let *nextValue* be ? **IteratorValue**(*next*).
 - d. Perform ! **CreateDataPropertyOrThrow**(*array*, ! **ToString**(**F**(*nextIndex*)), *nextValue*).
 - e. Set *nextIndex* to *nextIndex* + 1.

NOTE

CreateDataPropertyOrThrow 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]]**.

13.2.5.2 Runtime Semantics: Evaluation

ArrayLiteral : [*Elision*_{opt}]

1. Let *array* be ! **ArrayCreate**(0).
2. If *Elision* is present, then
 - a. Let *len* be the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and 0.
 - b. **ReturnIfAbrupt**(*len*).
3. 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*).
4. Return *array*.

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

1. Let *array* be ! **ArrayCreate**(0).
2. Let *nextIndex* be the result of performing **ArrayAccumulation** for *ElementList* with arguments *array* and 0.
3. **ReturnIfAbrupt**(*nextIndex*).
4. If *Elision* is present, then
 - a. Let *len* be the result of performing **ArrayAccumulation** for *Elision* with arguments *array* and *nextIndex*.
 - b. **ReturnIfAbrupt**(*len*).
5. Return *array*.

13.2.6 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 keys and associated values, enclosed in curly brackets. The values need not be literals; they are evaluated each time the object initializer is evaluated.

Syntax

```
ObjectLiteral[Yield, Await] :  
  { }  
  { PropertyDefinitionList[?Yield, ?Await] }  
  { PropertyDefinitionList[?Yield, ?Await] , }
```

```
PropertyDefinitionList[Yield, Await] :  
  PropertyDefinition[?Yield, ?Await]  
  PropertyDefinitionList[?Yield, ?Await] , PropertyDefinition[?Yield, ?Await]
```

```
PropertyDefinition[Yield, Await] :  
  IdentifierReference[?Yield, ?Await]  
  CoverInitializedName[?Yield, ?Await]  
  PropertyName[?Yield, ?Await] : AssignmentExpression[+In, ?Yield, ?Await]
```

*MethodDefinition*_[?Yield, ?Await]
... *AssignmentExpression*_[+In, ?Yield, ?Await]

*PropertyName*_[Yield, Await] :
LiteralPropertyName
*ComputedPropertyName*_[?Yield, ?Await]

LiteralPropertyName :
IdentifierName
StringLiteral
NumericLiteral

*ComputedPropertyName*_[Yield, Await] :
[*AssignmentExpression*_[+In, ?Yield, ?Await]]

*CoverInitializedName*_[Yield, Await] :
*IdentifierReference*_[?Yield, ?Await] *Initializer*_[+In, ?Yield, ?Await]

*Initializer*_[In, Yield, Await] :
= *AssignmentExpression*_[?In, ?Yield, ?Await]

NOTE 2 *MethodDefinition* is defined in 15.4.

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.

13.2.6.1 Static Semantics: Early Errors

PropertyDefinition : *MethodDefinition*

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

In addition to describing an actual object initializer the *ObjectLiteral* productions are also used as a cover grammar for *ObjectAssignmentPattern* 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* or *CoverCallExpressionAndAsyncArrowHead*.

PropertyDefinition : *CoverInitializedName*

- Always throw a Syntax Error if code matches this production.

NOTE This production exists so that *ObjectLiteral* can serve as a cover grammar for *ObjectAssignmentPattern*. It cannot occur in an actual object initializer.

13.2.6.2 Static Semantics: IsComputedPropertyKey

PropertyName : *LiteralPropertyName*

1. Return **false**.

PropertyName : *ComputedPropertyName*

1. Return **true**.

13.2.6.3 Static Semantics: PropertyNameList

PropertyDefinitionList : *PropertyDefinition*

1. If **PropName** of *PropertyDefinition* is **empty**, return a new empty **List**.
2. Return a **List** whose sole element is **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*.

13.2.6.4 Runtime Semantics: Evaluation

ObjectLiteral : { }

1. Return ! **OrdinaryObjectCreate**(%*Object.prototype*%).

ObjectLiteral :

{ *PropertyDefinitionList* }

{ *PropertyDefinitionList* , }

1. Let *obj* be ! **OrdinaryObjectCreate**(%*Object.prototype*%).
2. Perform ? **PropertyDefinitionEvaluation** of *PropertyDefinitionList* with arguments *obj* and **true**.
3. Return *obj*.

LiteralPropertyName : *IdentifierName*

1. Return **StringValue** of *IdentifierName*.

LiteralPropertyName : *StringLiteral*

1. Return the **SV** of *StringLiteral*.

LiteralPropertyName : *NumericLiteral*

1. Let *nbr* be the **NumericValue** of *NumericLiteral*.
2. Return ! **ToString**(*nbr*).

ComputedPropertyName : [*AssignmentExpression*]

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

13.2.6.5 Runtime Semantics: PropertyDefinitionEvaluation

With parameters *object* and *enumerable*.

PropertyDefinitionList : *PropertyDefinitionList* , *PropertyDefinition*

1. Perform ? [PropertyDefinitionEvaluation](#) of *PropertyDefinitionList* with arguments *object* and *enumerable*.
2. Return the result of performing [PropertyDefinitionEvaluation](#) of *PropertyDefinition* with arguments *object* and *enumerable*.

PropertyDefinition : ... *AssignmentExpression*

1. Let *exprValue* be the result of evaluating *AssignmentExpression*.
2. Let *fromValue* be ? [GetValue](#)(*exprValue*).
3. Let *excludedNames* be a new empty [List](#).
4. Return ? [CopyDataProperties](#)(*object*, *fromValue*, *excludedNames*).

PropertyDefinition : *IdentifierReference*

1. Let *propName* be [StringValue](#) of *IdentifierReference*.
2. Let *exprValue* be the result of evaluating *IdentifierReference*.
3. Let *propValue* be ? [GetValue](#)(*exprValue*).
4. **Assert**: *enumerable* is **true**.
5. **Assert**: *object* is an ordinary, extensible object with no non-configurable properties.
6. Return ! [CreateDataPropertyOrThrow](#)(*object*, *propName*, *propValue*).

PropertyDefinition : *PropertyName* : *AssignmentExpression*

1. Let *propKey* be the result of evaluating *PropertyName*.
2. [ReturnIfAbrupt](#)(*propKey*).
3. If [IsAnonymousFunctionDefinition](#)(*AssignmentExpression*) is **true**, then
 - a. Let *propValue* be ? [NamedEvaluation](#) of *AssignmentExpression* with argument *propKey*.
4. Else,
 - a. Let *exprValueRef* be the result of evaluating *AssignmentExpression*.
 - b. Let *propValue* be ? [GetValue](#)(*exprValueRef*).
5. **Assert**: *enumerable* is **true**.
6. **Assert**: *object* is an ordinary, extensible object with no non-configurable properties.
7. Return ! [CreateDataPropertyOrThrow](#)(*object*, *propKey*, *propValue*).

NOTE An alternative semantics for this production is given in [B.3.1](#).

MethodDefinition :

```
PropertyName ( UniqueFormalParameters ) { FunctionBody }  
get PropertyName ( ) { FunctionBody }  
set PropertyName ( PropertySetParameterList ) { FunctionBody }
```

1. Return ? [MethodDefinitionEvaluation](#) of *MethodDefinition* with arguments *object* and *enumerable*.

GeneratorMethod : * *PropertyName* (*UniqueFormalParameters*) { *GeneratorBody* }

1. Return ? [MethodDefinitionEvaluation](#) of *GeneratorMethod* with arguments *object* and *enumerable*.

AsyncGeneratorMethod : **async** * *PropertyName* (*UniqueFormalParameters*) { *AsyncGeneratorBody* }

1. Return ? [MethodDefinitionEvaluation](#) of *AsyncGeneratorMethod* with arguments *object* and *enumerable*.

AsyncMethod : **async** *PropertyName* (*UniqueFormalParameters*) { *AsyncFunctionBody* }

1. Return ? [MethodDefinitionEvaluation](#) of *AsyncMethod* with arguments *object* and *enumerable*.

13.2.7 Function Defining Expressions

See [15.2](#) for *PrimaryExpression* : *FunctionExpression* .

See [15.5](#) for *PrimaryExpression* : *GeneratorExpression* .

See [15.7](#) for *PrimaryExpression* : *ClassExpression* .

See [15.8](#) for *PrimaryExpression* : *AsyncFunctionExpression* .

See [15.6](#) for *PrimaryExpression* : *AsyncGeneratorExpression* .

13.2.8 Regular Expression Literals

Syntax

See [12.8.5](#).

13.2.8.1 Static Semantics: Early Errors

PrimaryExpression : *RegularExpressionLiteral*

- It is a Syntax Error if [IsValidRegularExpressionLiteral](#)(*RegularExpressionLiteral*) is **false**.

13.2.8.2 Static Semantics: [IsValidRegularExpressionLiteral](#) (*literal*)

The abstract operation [IsValidRegularExpressionLiteral](#) takes argument *literal*. It determines if its argument is a valid regular expression literal. It performs the following steps when called:

1. **Assert**: *literal* is a *RegularExpressionLiteral*.
2. If [FlagText](#) of *literal* contains any code points other than **g**, **i**, **m**, **s**, **u**, or **y**, or if it contains the same code point more than once, return **false**.
3. Let *patternText* be [BodyText](#) of *literal*.
4. If [FlagText](#) of *literal* contains **u**, let *u* be **true**; else let *u* be **false**.
5. If *u* is **false**, then
 - a. Let *stringValue* be [CodePointsToString](#)(*patternText*).
 - b. Set *patternText* to the sequence of code points resulting from interpreting each of the 16-bit elements of *stringValue* as a Unicode BMP code point. UTF-16 decoding is not applied to the elements.
6. Let *parseResult* be [ParsePattern](#)(*patternText*, *u*).
7. If *parseResult* is a [Parse Node](#), return **true**; else return **false**.

13.2.8.3 Runtime Semantics: Evaluation

PrimaryExpression : *RegularExpressionLiteral*

1. Let *pattern* be ! `CodePointsToString(BodyText of RegularExpressionLiteral)`.
2. Let *flags* be ! `CodePointsToString(FlagText of RegularExpressionLiteral)`.
3. Return `RegExpCreate(pattern, flags)`.

13.2.9 Template Literals

Syntax

`TemplateLiteral`_[Yield, Await, Tagged] :

`NoSubstitutionTemplate`

`SubstitutionTemplate`_[?Yield, ?Await, ?Tagged]

`SubstitutionTemplate`_[Yield, Await, Tagged] :

`TemplateHead` `Expression`_[+In, ?Yield, ?Await] `TemplateSpans`_[?Yield, ?Await, ?Tagged]

`TemplateSpans`_[Yield, Await, Tagged] :

`TemplateTail`

`TemplateMiddleList`_[?Yield, ?Await, ?Tagged] `TemplateTail`

`TemplateMiddleList`_[Yield, Await, Tagged] :

`TemplateMiddle` `Expression`_[+In, ?Yield, ?Await]

`TemplateMiddleList`_[?Yield, ?Await, ?Tagged] `TemplateMiddle` `Expression`_[+In, ?Yield, ?Await]

13.2.9.1 Static Semantics: Early Errors

`TemplateLiteral`_[Yield, Await, Tagged] : `NoSubstitutionTemplate`

- It is a Syntax Error if the [Tagged] parameter was not set and `NoSubstitutionTemplate` **Contains** `NotEscapeSequence`.

`TemplateLiteral`_[Yield, Await, Tagged] : `SubstitutionTemplate`_[?Yield, ?Await, ?Tagged]

- It is a Syntax Error if the number of elements in the result of `TemplateStrings` of `TemplateLiteral` with argument **false** is greater than $2^{32} - 1$.

`SubstitutionTemplate`_[Yield, Await, Tagged] : `TemplateHead` `Expression`_[+In, ?Yield, ?Await]

`TemplateSpans`_[?Yield, ?Await, ?Tagged]

- It is a Syntax Error if the [Tagged] parameter was not set and `TemplateHead` **Contains** `NotEscapeSequence`.

`TemplateSpans`_[Yield, Await, Tagged] : `TemplateTail`

- It is a Syntax Error if the [Tagged] parameter was not set and `TemplateTail` **Contains** `NotEscapeSequence`.

`TemplateMiddleList`_[Yield, Await, Tagged] :

`TemplateMiddle` `Expression`_[+In, ?Yield, ?Await]

`TemplateMiddleList`_[?Yield, ?Await, ?Tagged] `TemplateMiddle` `Expression`_[+In, ?Yield, ?Await]

- It is a Syntax Error if the [Tagged] parameter was not set and `TemplateMiddle` **Contains** `NotEscapeSequence`.

13.2.9.2 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** whose sole element is *string*.

SubstitutionTemplate : *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** whose elements are *head* followed by the elements 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** whose sole element is *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** whose elements are the elements 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** whose sole element is *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 *last* as the last element of the *List front*.
5. Return *front*.

13.2.9.3 GetTemplateObject (*templateLiteral*)

The abstract operation GetTemplateObject takes argument *templateLiteral* (a *Parse Node*). It performs the following steps when called:

1. Let *realm* be the current Realm Record.
2. Let *templateRegistry* be *realm*.[[TemplateMap]].
3. For each element *e* of *templateRegistry*, do
 - a. If *e*.[[Site]] is the same Parse Node as *templateLiteral*, then
 - i. Return *e*.[[Array]].
4. Let *rawStrings* be *TemplateStrings* of *templateLiteral* with argument **true**.
5. Let *cookedStrings* be *TemplateStrings* of *templateLiteral* with argument **false**.
6. Let *count* be the number of elements in the *List cookedStrings*.
7. **Assert:** $count \leq 2^{32} - 1$.
8. Let *template* be ! *ArrayCreate*(*count*).
9. Let *rawObj* be ! *ArrayCreate*(*count*).
10. Let *index* be 0.
11. Repeat, while *index* < *count*,
 - a. Let *prop* be ! *ToString*(*F*(*index*)).
 - b. Let *cookedValue* be *cookedStrings*[*index*].
 - c. Perform ! *DefinePropertyOrThrow*(*template*, *prop*, PropertyDescriptor { [[Value]]: *cookedValue*, [[Writable]]: **false**, [[Enumerable]]: **true**, [[Configurable]]: **false** }).
 - d. Let *rawValue* be the String value *rawStrings*[*index*].
 - e. Perform ! *DefinePropertyOrThrow*(*rawObj*, *prop*, PropertyDescriptor { [[Value]]: *rawValue*, [[Writable]]: **false**, [[Enumerable]]: **true**, [[Configurable]]: **false** }).
 - f. Set *index* to *index* + 1.
12. Perform ! *SetIntegrityLevel*(*rawObj*, frozen).
13. Perform ! *DefinePropertyOrThrow*(*template*, "raw", PropertyDescriptor { [[Value]]: *rawObj*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
14. Perform ! *SetIntegrityLevel*(*template*, frozen).
15. Append the *Record* { [[Site]]: *templateLiteral*, [[Array]]: *template* } to *templateRegistry*.
16. Return *template*.

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

NOTE 2 Each *TemplateLiteral* in the program code of a *realm* is associated with a unique template object that is used in the evaluation of tagged Templates (13.2.9.5). The template objects are frozen and the same template object is used each time a specific tagged Template is evaluated. Whether template 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 template objects.

13.2.9.4 Runtime Semantics: SubstitutionEvaluation

TemplateSpans : *TemplateTail*

1. Return a new empty [List](#).

TemplateSpans : *TemplateMiddleList* *TemplateTail*

1. Return the result of [SubstitutionEvaluation](#) of *TemplateMiddleList*.

TemplateMiddleList : *TemplateMiddle* *Expression*

1. Let *subRef* be the result of evaluating *Expression*.
2. Let *sub* be ? [GetValue](#)(*subRef*).
3. Return a [List](#) whose sole element is *sub*.

TemplateMiddleList : *TemplateMiddleList* *TemplateMiddle* *Expression*

1. Let *preceding* be ? [SubstitutionEvaluation](#) of *TemplateMiddleList*.
2. Let *nextRef* be the result of evaluating *Expression*.
3. Let *next* be ? [GetValue](#)(*nextRef*).
4. Append *next* as the last element of the [List](#) *preceding*.
5. Return *preceding*.

13.2.9.5 Runtime Semantics: Evaluation

TemplateLiteral : *NoSubstitutionTemplate*

1. Return the TV of *NoSubstitutionTemplate* as defined in [12.8.6](#).

SubstitutionTemplate : *TemplateHead* *Expression* *TemplateSpans*

1. Let *head* be the TV of *TemplateHead* as defined in [12.8.6](#).
2. Let *subRef* be the result of evaluating *Expression*.
3. Let *sub* be ? [GetValue](#)(*subRef*).
4. Let *middle* be ? [ToString](#)(*sub*).
5. Let *tail* be the result of evaluating *TemplateSpans*.
6. [ReturnIfAbrupt](#)(*tail*).
7. Return the [string-concatenation](#) of *head*, *middle*, and *tail*.

NOTE 1 The string conversion semantics applied to the *Expression* value are like **String.prototype.concat** rather than the **+** operator.

TemplateSpans : *TemplateTail*

1. Return the TV of *TemplateTail* as defined in [12.8.6](#).

TemplateSpans : *TemplateMiddleList* *TemplateTail*

1. Let *head* be the result of evaluating *TemplateMiddleList*.
2. **ReturnIfAbrupt**(*head*).
3. Let *tail* be the TV of *TemplateTail* as defined in 12.8.6.
4. Return the **string-concatenation** of *head* and *tail*.

TemplateMiddleList : *TemplateMiddle Expression*

1. Let *head* be the TV of *TemplateMiddle* as defined in 12.8.6.
2. Let *subRef* be the result of evaluating *Expression*.
3. Let *sub* be ? **GetValue**(*subRef*).
4. Let *middle* be ? **Tostring**(*sub*).
5. Return the **string-concatenation** of *head* and *middle*.

NOTE 2 The string conversion semantics applied to the *Expression* value are like **String.prototype.concat** rather than the **+** operator.

TemplateMiddleList : *TemplateMiddleList TemplateMiddle Expression*

1. Let *rest* be the result of evaluating *TemplateMiddleList*.
2. **ReturnIfAbrupt**(*rest*).
3. Let *middle* be the TV of *TemplateMiddle* as defined in 12.8.6.
4. Let *subRef* be the result of evaluating *Expression*.
5. Let *sub* be ? **GetValue**(*subRef*).
6. Let *last* be ? **Tostring**(*sub*).
7. Return the **string-concatenation** of *rest*, *middle*, and *last*.

NOTE 3 The string conversion semantics applied to the *Expression* value are like **String.prototype.concat** rather than the **+** operator.

13.2.10 The Grouping Operator

13.2.10.1 Static Semantics: Early Errors

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

- It is a Syntax Error if *CoverParenthesizedExpressionAndArrowParameterList* is not **covering** a *ParenthesizedExpression*.
- All Early Error rules for *ParenthesizedExpression* and its derived productions also apply to **CoveredParenthesizedExpression** of *CoverParenthesizedExpressionAndArrowParameterList*.

13.2.10.2 Runtime Semantics: Evaluation

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

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

ParenthesizedExpression : (*Expression*)

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

NOTE

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

13.3 Left-Hand-Side Expressions

Syntax

```

MemberExpression[Yield, Await] :
    PrimaryExpression[?Yield, ?Await]
    MemberExpression[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    MemberExpression[?Yield, ?Await] . IdentifierName
    MemberExpression[?Yield, ?Await] TemplateLiteral[?Yield, ?Await, +Tagged]
    SuperProperty[?Yield, ?Await]
    MetaProperty
    new MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]

SuperProperty[Yield, Await] :
    super [ Expression[+In, ?Yield, ?Await] ]
    super . IdentifierName

MetaProperty :
    NewTarget
    ImportMeta

NewTarget :
    new . target

ImportMeta :
    import . meta

NewExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await]
    new NewExpression[?Yield, ?Await]

CallExpression[Yield, Await] :
    CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await]
    SuperCall[?Yield, ?Await]
    ImportCall[?Yield, ?Await]
    CallExpression[?Yield, ?Await] Arguments[?Yield, ?Await]
    CallExpression[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    CallExpression[?Yield, ?Await] . IdentifierName
    CallExpression[?Yield, ?Await] TemplateLiteral[?Yield, ?Await, +Tagged]

```

```

SuperCall[Yield, Await] :
    super Arguments[?Yield, ?Await]

ImportCall[Yield, Await] :
    import ( AssignmentExpression[+In, ?Yield, ?Await] )

Arguments[Yield, Await] :
    ( )
    ( ArgumentList[?Yield, ?Await] )
    ( ArgumentList[?Yield, ?Await] , )

ArgumentList[Yield, Await] :
    AssignmentExpression[+In, ?Yield, ?Await]
    ... AssignmentExpression[+In, ?Yield, ?Await]
    ArgumentList[?Yield, ?Await] , AssignmentExpression[+In, ?Yield, ?Await]
    ArgumentList[?Yield, ?Await] , ... AssignmentExpression[+In, ?Yield, ?Await]

OptionalExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await] OptionalChain[?Yield, ?Await]
    CallExpression[?Yield, ?Await] OptionalChain[?Yield, ?Await]
    OptionalExpression[?Yield, ?Await] OptionalChain[?Yield, ?Await]

OptionalChain[Yield, Await] :
    ?. Arguments[?Yield, ?Await]
    ?. [ Expression[+In, ?Yield, ?Await] ]
    ?. IdentifierName
    ?. TemplateLiteral[?Yield, ?Await, +Tagged]
    OptionalChain[?Yield, ?Await] Arguments[?Yield, ?Await]
    OptionalChain[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    OptionalChain[?Yield, ?Await] . IdentifierName
    OptionalChain[?Yield, ?Await] TemplateLiteral[?Yield, ?Await, +Tagged]

LeftHandSideExpression[Yield, Await] :
    NewExpression[?Yield, ?Await]
    CallExpression[?Yield, ?Await]
    OptionalExpression[?Yield, ?Await]

```

Supplemental Syntax

When processing an instance of the production

CallExpression : *CoverCallExpressionAndAsyncArrowHead*

the interpretation of *CoverCallExpressionAndAsyncArrowHead* is refined using the following grammar:

```

CallMemberExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]

```

13.3.1 Static Semantics

13.3.1.1 Static Semantics: Early Errors

OptionalChain :

? . TemplateLiteral

OptionalChain TemplateLiteral

- It is a Syntax Error if any code matches this production.

NOTE

This production exists in order to prevent automatic semicolon insertion rules (12.9) from being applied to the following code:

```
a?.b  
`c`
```

so that it would be interpreted as two valid statements. The purpose is to maintain consistency with similar code without optional chaining:

```
a.b  
`c`
```

which is a valid statement and where automatic semicolon insertion does not apply.

ImportMeta :

```
import . meta
```

- It is a Syntax Error if the syntactic [goal symbol](#) is not *Module*.

13.3.1.2 Static Semantics: CoveredCallExpression

CoverCallExpressionAndAsyncArrowHead : *MemberExpression Arguments*

1. Return the *CallMemberExpression* that is [covered](#) by *CoverCallExpressionAndAsyncArrowHead*.

13.3.2 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*.

13.3.2.1 Runtime Semantics: Evaluation

MemberExpression : *MemberExpression* [*Expression*]

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be ? [GetValue](#)(*baseReference*).
3. If the code matched by this *MemberExpression* is [strict mode code](#), let *strict* be **true**; else let *strict* be **false**.
4. Return ? [EvaluatePropertyAccessWithExpressionKey](#)(*baseValue*, *Expression*, *strict*).

MemberExpression : *MemberExpression* . *IdentifierName*

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be ? [GetValue](#)(*baseReference*).
3. If the code matched by this *MemberExpression* is [strict mode code](#), let *strict* be **true**; else let *strict* be **false**.
4. Return ? [EvaluatePropertyAccessWithIdentifierKey](#)(*baseValue*, *IdentifierName*, *strict*).

CallExpression : *CallExpression* [*Expression*]

1. Let *baseReference* be the result of evaluating *CallExpression*.
2. Let *baseValue* be ? [GetValue](#)(*baseReference*).
3. If the code matched by this *CallExpression* is [strict mode code](#), let *strict* be **true**; else let *strict* be **false**.
4. Return ? [EvaluatePropertyAccessWithExpressionKey](#)(*baseValue*, *Expression*, *strict*).

CallExpression : *CallExpression* . *IdentifierName*

1. Let *baseReference* be the result of evaluating *CallExpression*.
2. Let *baseValue* be ? `GetValue`(*baseReference*).
3. If the code matched by this *CallExpression* is **strict mode code**, let *strict* be **true**; else let *strict* be **false**.
4. Return ? `EvaluatePropertyAccessWithIdentifierKey`(*baseValue*, *IdentifierName*, *strict*).

13.3.3 EvaluatePropertyAccessWithExpressionKey (*baseValue*, *expression*, *strict*)

The abstract operation `EvaluatePropertyAccessWithExpressionKey` takes arguments *baseValue* (an **ECMAScript language value**), *expression* (a **Parse Node**), and *strict* (a Boolean). It performs the following steps when called:

1. Let *propertyNameReference* be the result of evaluating *expression*.
2. Let *propertyNameValue* be ? `GetValue`(*propertyNameReference*).
3. Let *bv* be ? `RequireObjectCoercible`(*baseValue*).
4. Let *propertyKey* be ? `ToPropertyKey`(*propertyNameValue*).
5. Return the **Reference Record** { `[[Base]]`: *bv*, `[[ReferencedName]]`: *propertyKey*, `[[Strict]]`: *strict*, `[[ThisValue]]`: empty }.

13.3.4 EvaluatePropertyAccessWithIdentifierKey (*baseValue*, *identifierName*, *strict*)

The abstract operation `EvaluatePropertyAccessWithIdentifierKey` takes arguments *baseValue* (an **ECMAScript language value**), *identifierName* (a **Parse Node**), and *strict* (a Boolean). It performs the following steps when called:

1. **Assert**: *identifierName* is an *IdentifierName*.
2. Let *bv* be ? `RequireObjectCoercible`(*baseValue*).
3. Let *propertyNameString* be **StringValue** of *identifierName*.
4. Return the **Reference Record** { `[[Base]]`: *bv*, `[[ReferencedName]]`: *propertyNameString*, `[[Strict]]`: *strict*, `[[ThisValue]]`: empty }.

13.3.5 The new Operator

13.3.5.1 Runtime Semantics: Evaluation

NewExpression : **new** *NewExpression*

1. Return ? `EvaluateNew`(*NewExpression*, empty).

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

1. Return ? `EvaluateNew`(*MemberExpression*, *Arguments*).

13.3.5.1.1 EvaluateNew (*constructExpr*, *arguments*)

The abstract operation `EvaluateNew` takes arguments *constructExpr* and *arguments*. It performs the following steps when called:

1. **Assert**: *constructExpr* is either a *NewExpression* or a *MemberExpression*.
2. **Assert**: *arguments* is either **empty** or an *Arguments*.
3. Let *ref* be the result of evaluating *constructExpr*.
4. Let *constructor* be ? `GetValue`(*ref*).
5. If *arguments* is **empty**, let *argList* be a new empty **List**.

6. Else,
 - a. Let *argList* be ? [ArgumentListEvaluation](#) of *arguments*.
7. If [IsConstructor](#)(*constructor*) is **false**, throw a **TypeError** exception.
8. Return ? [Construct](#)(*constructor*, *argList*).

13.3.6 Function Calls

13.3.6.1 Runtime Semantics: Evaluation

CallExpression : *CoverCallExpressionAndAsyncArrowHead*

1. Let *expr* be [CoveredCallExpression](#) of *CoverCallExpressionAndAsyncArrowHead*.
2. Let *memberExpr* be the *MemberExpression* of *expr*.
3. Let *arguments* be the *Arguments* of *expr*.
4. Let *ref* be the result of evaluating *memberExpr*.
5. Let *func* be ? [GetValue](#)(*ref*).
6. If *ref* is a [Reference Record](#), [IsPropertyReference](#)(*ref*) is **false**, and *ref*.[[ReferencedName]] is "eval", then
 - a. If [SameValue](#)(*func*, %eval%) is **true**, then
 - i. Let *argList* be ? [ArgumentListEvaluation](#) of *arguments*.
 - ii. If *argList* has no elements, return **undefined**.
 - iii. Let *evalArg* be the first element of *argList*.
 - iv. If the source code matching this *CallExpression* is [strict mode code](#), let *strictCaller* be **true**. Otherwise let *strictCaller* be **false**.
 - v. Let *evalRealm* be the [current Realm Record](#).
 - vi. Return ? [PerformEval](#)(*evalArg*, *evalRealm*, *strictCaller*, **true**).
7. Let *thisCall* be this *CallExpression*.
8. Let *tailCall* be [IsInTailPosition](#)(*thisCall*).
9. Return ? [EvaluateCall](#)(*func*, *ref*, *arguments*, *tailCall*).

A *CallExpression* evaluation that executes step 6.a.vi is a *direct eval*.

CallExpression : *CallExpression Arguments*

1. Let *ref* be the result of evaluating *CallExpression*.
2. Let *func* be ? [GetValue](#)(*ref*).
3. Let *thisCall* be this *CallExpression*.
4. Let *tailCall* be [IsInTailPosition](#)(*thisCall*).
5. Return ? [EvaluateCall](#)(*func*, *ref*, *Arguments*, *tailCall*).

13.3.6.2 EvaluateCall (*func*, *ref*, *arguments*, *tailPosition*)

The abstract operation [EvaluateCall](#) takes arguments *func* (an [ECMAScript language value](#)), *ref* (an [ECMAScript language value](#) or a [Reference Record](#)), *arguments* (a [Parse Node](#)), and *tailPosition* (a Boolean). It performs the following steps when called:

1. If *ref* is a [Reference Record](#), then
 - a. If [IsPropertyReference](#)(*ref*) is **true**, then
 - i. Let *thisValue* be [GetThisValue](#)(*ref*).
 - b. Else,
 - i. Let *refEnv* be *ref*.[[Base]].

- ii. **Assert:** *refEnv* is an **Environment Record**.
 - iii. Let *thisValue* be *refEnv*.WithBaseObject().
2. Else,
 - a. Let *thisValue* be **undefined**.
 3. Let *argList* be ? **ArgumentListEvaluation** of *arguments*.
 4. If **Type**(*func*) is not **Object**, throw a **TypeError** exception.
 5. If **IsCallable**(*func*) is **false**, throw a **TypeError** exception.
 6. If *tailPosition* is **true**, perform **PrepareForTailCall**().
 7. Let *result* be **Call**(*func*, *thisValue*, *argList*).
 8. **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.
 9. **Assert:** If *result* is not an **abrupt completion**, then **Type**(*result*) is an **ECMAScript language type**.
 10. Return *result*.

13.3.7 The **super** Keyword

13.3.7.1 Runtime Semantics: Evaluation

SuperProperty : **super** [*Expression*]

1. Let *env* be **GetThisEnvironment**().
2. Let *actualThis* be ? *env*.**GetThisBinding**().
3. Let *propertyNameReference* be the result of evaluating *Expression*.
4. Let *propertyNameValue* be ? **GetValue**(*propertyNameReference*).
5. Let *propertyKey* be ? **ToPropertyKey**(*propertyNameValue*).
6. If the code matched by this *SuperProperty* is **strict mode code**, let *strict* be **true**; else let *strict* be **false**.
7. Return ? **MakeSuperPropertyReference**(*actualThis*, *propertyKey*, *strict*).

SuperProperty : **super** . *IdentifierName*

1. Let *env* be **GetThisEnvironment**().
2. Let *actualThis* be ? *env*.**GetThisBinding**().
3. Let *propertyKey* be **StringValue** of *IdentifierName*.
4. If the code matched by this *SuperProperty* is **strict mode code**, let *strict* be **true**; else let *strict* be **false**.
5. Return ? **MakeSuperPropertyReference**(*actualThis*, *propertyKey*, *strict*).

SuperCall : **super** *Arguments*

1. Let *newTarget* be **GetNewTarget**().
2. **Assert:** **Type**(*newTarget*) is **Object**.
3. Let *func* be ! **GetSuperConstructor**().
4. Let *argList* be ? **ArgumentListEvaluation** of *Arguments*.
5. If **IsConstructor**(*func*) is **false**, throw a **TypeError** exception.
6. Let *result* be ? **Construct**(*func*, *argList*, *newTarget*).
7. Let *thisER* be **GetThisEnvironment**().
8. Return ? *thisER*.**BindThisValue**(*result*).

13.3.7.2 **GetSuperConstructor** ()

The abstract operation **GetSuperConstructor** takes no arguments. It performs the following steps when called:

1. Let *envRec* be `GetThisEnvironment()`.
2. **Assert:** *envRec* is a function `Environment Record`.
3. Let *activeFunction* be *envRec*.`[[FunctionObject]]`.
4. **Assert:** *activeFunction* is an ECMAScript `function object`.
5. Let *superConstructor* be `! activeFunction. [[GetPrototypeOf]]()`.
6. Return *superConstructor*.

13.3.7.3 MakeSuperPropertyReference (*actualThis*, *propertyKey*, *strict*)

The abstract operation `MakeSuperPropertyReference` takes arguments *actualThis*, *propertyKey*, and *strict*. It performs the following steps when called:

1. Let *env* be `GetThisEnvironment()`.
2. **Assert:** *env*.`HasSuperBinding()` is **true**.
3. Let *baseValue* be `? env.GetSuperBase()`.
4. Let *bv* be `? RequireObjectCoercible(baseValue)`.
5. Return the `Reference Record` { `[[Base]]`: *bv*, `[[ReferencedName]]`: *propertyKey*, `[[Strict]]`: *strict*, `[[ThisValue]]`: *actualThis* }.
6. NOTE: This returns a `Super Reference Record`.

13.3.8 Argument Lists

NOTE The evaluation of an argument list produces a `List` of values.

13.3.8.1 Runtime Semantics: ArgumentListEvaluation

Arguments : ()

1. Return a new empty `List`.

ArgumentList : *AssignmentExpression*

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be `? GetValue(ref)`.
3. Return a `List` whose sole element is *arg*.

ArgumentList : ... *AssignmentExpression*

1. Let *list* be a new empty `List`.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadObj* be `? GetValue(spreadRef)`.
4. Let *iteratorRecord* be `? GetIterator(spreadObj)`.
5. Repeat,
 - a. Let *next* be `? IteratorStep(iteratorRecord)`.
 - b. If *next* is **false**, return *list*.
 - c. Let *nextArg* be `? IteratorValue(next)`.
 - d. Append *nextArg* as the last element of *list*.

ArgumentList : *ArgumentList* , *AssignmentExpression*

1. Let *precedingArgs* be ? [ArgumentListEvaluation](#) of *ArgumentList*.
2. Let *ref* be the result of evaluating *AssignmentExpression*.
3. Let *arg* be ? [GetValue](#)(*ref*).
4. Append *arg* to the end of *precedingArgs*.
5. Return *precedingArgs*.

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

1. Let *precedingArgs* be ? [ArgumentListEvaluation](#) of *ArgumentList*.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *iteratorRecord* be ? [GetIterator](#)(? [GetValue](#)(*spreadRef*)).
4. Repeat,
 - a. Let *next* be ? [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is **false**, return *precedingArgs*.
 - c. Let *nextArg* be ? [IteratorValue](#)(*next*).
 - d. Append *nextArg* as the last element of *precedingArgs*.

TemplateLiteral : *NoSubstitutionTemplate*

1. Let *templateLiteral* be this *TemplateLiteral*.
2. Let *siteObj* be [GetTemplateObject](#)(*templateLiteral*).
3. Return a [List](#) whose sole element is *siteObj*.

TemplateLiteral : *SubstitutionTemplate*

1. Let *templateLiteral* be this *TemplateLiteral*.
2. Let *siteObj* be [GetTemplateObject](#)(*templateLiteral*).
3. Let *remaining* be ? [ArgumentListEvaluation](#) of *SubstitutionTemplate*.
4. Return a [List](#) whose first element is *siteObj* and whose subsequent elements are the elements of *remaining*.

SubstitutionTemplate : *TemplateHead* *Expression* *TemplateSpans*

1. Let *firstSubRef* be the result of evaluating *Expression*.
2. Let *firstSub* be ? [GetValue](#)(*firstSubRef*).
3. Let *restSub* be ? [SubstitutionEvaluation](#) of *TemplateSpans*.
4. **Assert**: *restSub* is a [List](#).
5. Return a [List](#) whose first element is *firstSub* and whose subsequent elements are the elements of *restSub*. *restSub* may contain no elements.

13.3.9 Optional Chains

NOTE An optional chain is a chain of one or more property accesses and function calls, the first of which begins with the token ? ..

13.3.9.1 Runtime Semantics: Evaluation

OptionalExpression :

MemberExpression *OptionalChain*

1. Let *baseReference* be the result of evaluating *MemberExpression*.

2. Let *baseValue* be ? `GetValue(baseReference)`.
3. If *baseValue* is **undefined** or **null**, then
 - a. Return **undefined**.
4. Return the result of performing `ChainEvaluation` of *OptionalChain* with arguments *baseValue* and *baseReference*.

OptionalExpression :

CallExpression *OptionalChain*

1. Let *baseReference* be the result of evaluating *CallExpression*.
2. Let *baseValue* be ? `GetValue(baseReference)`.
3. If *baseValue* is **undefined** or **null**, then
 - a. Return **undefined**.
4. Return the result of performing `ChainEvaluation` of *OptionalChain* with arguments *baseValue* and *baseReference*.

OptionalExpression :

OptionalExpression *OptionalChain*

1. Let *baseReference* be the result of evaluating *OptionalExpression*.
2. Let *baseValue* be ? `GetValue(baseReference)`.
3. If *baseValue* is **undefined** or **null**, then
 - a. Return **undefined**.
4. Return the result of performing `ChainEvaluation` of *OptionalChain* with arguments *baseValue* and *baseReference*.

13.3.9.2 Runtime Semantics: ChainEvaluation

With parameters *baseValue* and *baseReference*.

OptionalChain : ?. *Arguments*

1. Let *thisChain* be this *OptionalChain*.
2. Let *tailCall* be `IsInTailPosition(thisChain)`.
3. Return ? `EvaluateCall(baseValue, baseReference, Arguments, tailCall)`.

OptionalChain : ?. [*Expression*]

1. If the code matched by this *OptionalChain* is **strict mode code**, let *strict* be **true**; else let *strict* be **false**.
2. Return ? `EvaluatePropertyAccessWithExpressionKey(baseValue, Expression, strict)`.

OptionalChain : ?. *IdentifierName*

1. If the code matched by this *OptionalChain* is **strict mode code**, let *strict* be **true**; else let *strict* be **false**.
2. Return ? `EvaluatePropertyAccessWithIdentifierKey(baseValue, IdentifierName, strict)`.

OptionalChain : *OptionalChain* *Arguments*

1. Let *optionalChain* be *OptionalChain*.
2. Let *newReference* be ? `ChainEvaluation` of *optionalChain* with arguments *baseValue* and *baseReference*.
3. Let *newValue* be ? `GetValue(newReference)`.
4. Let *thisChain* be this *OptionalChain*.
5. Let *tailCall* be `IsInTailPosition(thisChain)`.
6. Return ? `EvaluateCall(newValue, newReference, Arguments, tailCall)`.

OptionalChain : *OptionalChain* [*Expression*]

1. Let *optionalChain* be *OptionalChain*.
2. Let *newReference* be ? [ChainEvaluation](#) of *optionalChain* with arguments *baseValue* and *baseReference*.
3. Let *newValue* be ? [GetValue](#)(*newReference*).
4. If the code matched by this *OptionalChain* is [strict mode code](#), let *strict* be **true**; else let *strict* be **false**.
5. Return ? [EvaluatePropertyAccessWithExpressionKey](#)(*newValue*, *Expression*, *strict*).

OptionalChain : *OptionalChain* . *IdentifierName*

1. Let *optionalChain* be *OptionalChain*.
2. Let *newReference* be ? [ChainEvaluation](#) of *optionalChain* with arguments *baseValue* and *baseReference*.
3. Let *newValue* be ? [GetValue](#)(*newReference*).
4. If the code matched by this *OptionalChain* is [strict mode code](#), let *strict* be **true**; else let *strict* be **false**.
5. Return ? [EvaluatePropertyAccessWithIdentifierKey](#)(*newValue*, *IdentifierName*, *strict*).

13.3.10 Import Calls

13.3.10.1 Runtime Semantics: Evaluation

ImportCall : **import** (*AssignmentExpression*)

1. Let *referencingScriptOrModule* be ! [GetActiveScriptOrModule](#)().
2. Let *argRef* be the result of evaluating *AssignmentExpression*.
3. Let *specifier* be ? [GetValue](#)(*argRef*).
4. Let *promiseCapability* be ! [NewPromiseCapability](#)(%Promise%).
5. Let *specifierString* be [ToString](#)(*specifier*).
6. [IfAbruptRejectPromise](#)(*specifierString*, *promiseCapability*).
7. Perform ! [HostImportModuleDynamically](#)(*referencingScriptOrModule*, *specifierString*, *promiseCapability*).
8. Return *promiseCapability*.[[Promise]].

13.3.11 Tagged Templates

NOTE A tagged template is a function call where the arguments of the call are derived from a *TemplateLiteral* (13.2.9). The actual arguments include a template object (13.2.9.3) and the values produced by evaluating the expressions embedded within the *TemplateLiteral*.

13.3.11.1 Runtime Semantics: Evaluation

MemberExpression : *MemberExpression* *TemplateLiteral*

1. Let *tagRef* be the result of evaluating *MemberExpression*.
2. Let *tagFunc* be ? [GetValue](#)(*tagRef*).
3. Let *thisCall* be this *MemberExpression*.
4. Let *tailCall* be [IsInTailPosition](#)(*thisCall*).
5. Return ? [EvaluateCall](#)(*tagFunc*, *tagRef*, *TemplateLiteral*, *tailCall*).

CallExpression : *CallExpression* *TemplateLiteral*

1. Let *tagRef* be the result of evaluating *CallExpression*.

2. Let *tagFunc* be ? [GetValue](#)(*tagRef*).
3. Let *thisCall* be this *CallExpression*.
4. Let *tailCall* be [IsInTailPosition](#)(*thisCall*).
5. Return ? [EvaluateCall](#)(*tagFunc*, *tagRef*, *TemplateLiteral*, *tailCall*).

13.3.12 Meta Properties

13.3.12.1 Runtime Semantics: Evaluation

NewTarget : **new** . **target**

1. Return [GetNewTarget](#)().

ImportMeta : **import** . **meta**

1. Let *module* be ! [GetActiveScriptOrModule](#)().
2. **Assert**: *module* is a [Source Text Module Record](#).
3. Let *importMeta* be *module*.[[[ImportMeta](#)]].
4. If *importMeta* is empty, then
 - a. Set *importMeta* to ! [OrdinaryObjectCreate](#)(**null**).
 - b. Let *importMetaValues* be ! [HostGetImportMetaProperties](#)(*module*).
 - c. For each [Record](#) { [[[Key](#)]], [[[Value](#)]] } *p* of *importMetaValues*, do
 - i. Perform ! [CreateDataPropertyOrThrow](#)(*importMeta*, *p*.[[[Key](#)]], *p*.[[[Value](#)]]).
 - d. Perform ! [HostFinalizeImportMeta](#)(*importMeta*, *module*).
 - e. Set *module*.[[[ImportMeta](#)]] to *importMeta*.
 - f. Return *importMeta*.
5. Else,
 - a. **Assert**: [Type](#)(*importMeta*) is [Object](#).
 - b. Return *importMeta*.

13.3.12.1.1 [HostGetImportMetaProperties](#) (*moduleRecord*)

The [host-defined](#) abstract operation [HostGetImportMetaProperties](#) takes argument *moduleRecord* (a [Module Record](#)). It allows hosts to provide property keys and values for the object returned from **import.meta**.

The implementation of [HostGetImportMetaProperties](#) must conform to the following requirements:

- It must return a [List](#), whose values are all [Records](#) with two fields, [[[Key](#)]] and [[[Value](#)]].
- Each such [Record](#)'s [[[Key](#)]] field must be a property key, i.e., [IsPropertyKey](#) must return **true** when applied to it.
- Each such [Record](#)'s [[[Value](#)]] field must be an ECMAScript value.
- It must always complete normally (i.e., not return an [abrupt completion](#)).

The default implementation of [HostGetImportMetaProperties](#) is to return a new empty [List](#).

13.3.12.1.2 [HostFinalizeImportMeta](#) (*importMeta*, *moduleRecord*)

The [host-defined](#) abstract operation [HostFinalizeImportMeta](#) takes arguments *importMeta* (an [Object](#)) and *moduleRecord* (a [Module Record](#)). It allows hosts to perform any extraordinary operations to prepare the object returned from **import.meta**.

Most hosts will be able to simply define [HostGetImportMetaProperties](#), and leave `HostFinalizeImportMeta` with its default behaviour. However, `HostFinalizeImportMeta` provides an "escape hatch" for hosts which need to directly manipulate the object before it is exposed to ECMAScript code.

The implementation of `HostFinalizeImportMeta` must conform to the following requirements:

- It must always complete normally (i.e., not return an [abrupt completion](#)).

The default implementation of `HostFinalizeImportMeta` is to return [NormalCompletion](#)(empty).

13.4 Update Expressions

Syntax

```
UpdateExpression[Yield, Await] :  
  LeftHandSideExpression[?Yield, ?Await]  
  LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] ++  
  LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] --  
  ++ UnaryExpression[?Yield, ?Await]  
  -- UnaryExpression[?Yield, ?Await]
```

13.4.1 Static Semantics: Early Errors

UpdateExpression :

```
LeftHandSideExpression ++  
LeftHandSideExpression --
```

- It is an early Syntax Error if [AssignmentTargetType](#) of *LeftHandSideExpression* is not simple.

UpdateExpression :

```
++ UnaryExpression  
-- UnaryExpression
```

- It is an early Syntax Error if [AssignmentTargetType](#) of *UnaryExpression* is not simple.

13.4.2 Postfix Increment Operator

13.4.2.1 Runtime Semantics: Evaluation

UpdateExpression : *LeftHandSideExpression* ++

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be ? [ToNumeric](#)(? [GetValue](#)(*lhs*)).
3. Let *newValue* be ! [Type](#)(*oldValue*)::[add](#)(*oldValue*, [Type](#)(*oldValue*)::unit).
4. Perform ? [PutValue](#)(*lhs*, *newValue*).
5. Return *oldValue*.

13.4.3 Postfix Decrement Operator

13.4.3.1 Runtime Semantics: Evaluation

UpdateExpression : *LeftHandSideExpression* --

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be ? **ToNumeric**(? **GetValue**(*lhs*)).
3. Let *newValue* be ! **Type**(*oldValue*)::subtract(*oldValue*, **Type**(*oldValue*)::unit).
4. Perform ? **PutValue**(*lhs*, *newValue*).
5. Return *oldValue*.

13.4.4 Prefix Increment Operator

13.4.4.1 Runtime Semantics: Evaluation

UpdateExpression : ++ *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ? **ToNumeric**(? **GetValue**(*expr*)).
3. Let *newValue* be ! **Type**(*oldValue*)::add(*oldValue*, **Type**(*oldValue*)::unit).
4. Perform ? **PutValue**(*expr*, *newValue*).
5. Return *newValue*.

13.4.5 Prefix Decrement Operator

13.4.5.1 Runtime Semantics: Evaluation

UpdateExpression : -- *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ? **ToNumeric**(? **GetValue**(*expr*)).
3. Let *newValue* be ! **Type**(*oldValue*)::subtract(*oldValue*, **Type**(*oldValue*)::unit).
4. Perform ? **PutValue**(*expr*, *newValue*).
5. Return *newValue*.

13.5 Unary Operators

Syntax

```
UnaryExpression [Yield, Await] :  
  UpdateExpression [?Yield, ?Await]  
  delete UnaryExpression [?Yield, ?Await]  
  void UnaryExpression [?Yield, ?Await]  
  typeof UnaryExpression [?Yield, ?Await]  
  + UnaryExpression [?Yield, ?Await]  
  - UnaryExpression [?Yield, ?Await]  
  ~ UnaryExpression [?Yield, ?Await]  
  ! UnaryExpression [?Yield, ?Await]  
  [+Await] AwaitExpression [?Yield]
```

13.5.1 The **delete** Operator

13.5.1.1 Static Semantics: Early Errors

UnaryExpression : **delete** *UnaryExpression*

- It is a Syntax Error if the *UnaryExpression* is contained in **strict mode 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.

13.5.1.2 Runtime Semantics: Evaluation

UnaryExpression : **delete** *UnaryExpression*

1. Let *ref* be the result of evaluating *UnaryExpression*.
2. **ReturnIfAbrupt**(*ref*).
3. If *ref* is not a **Reference Record**, return **true**.
4. If **IsUnresolvableReference**(*ref*) is **true**, then
 - a. **Assert**: *ref*.[[Strict]] is **false**.
 - b. Return **true**.
5. If **IsPropertyReference**(*ref*) is **true**, then
 - a. If **IsSuperReference**(*ref*) is **true**, throw a **ReferenceError** exception.
 - b. Let *baseObj* be ! **ToObject**(*ref*.[[Base]]).
 - c. Let *deleteStatus* be ? *baseObj*.[[Delete]](*ref*.[[ReferencedName]]).
 - d. If *deleteStatus* is **false** and *ref*.[[Strict]] is **true**, throw a **TypeError** exception.
 - e. Return *deleteStatus*.
6. Else,
 - a. Let *base* be *ref*.[[Base]].
 - b. **Assert**: *base* is an **Environment Record**.
 - c. Return ? *base*.DeleteBinding(*ref*.[[ReferencedName]]).

NOTE 1 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** } (or otherwise cannot be deleted), a **TypeError** exception is thrown.

NOTE 2 The object that may be created in step 5.b is not accessible outside of the above abstract operation and the **ordinary object** [[Delete]] internal method. An implementation might choose to avoid the actual creation of that object.

13.5.2 The `void` Operator

13.5.2.1 Runtime Semantics: Evaluation

UnaryExpression : **void** *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Perform ? `GetValue(expr)`.
3. Return **undefined**.

NOTE `GetValue` must be called even though its value is not used because it may have observable side-effects.

13.5.3 The `typeof` Operator

13.5.3.1 Runtime Semantics: Evaluation

UnaryExpression : **typeof** *UnaryExpression*

1. Let *val* be the result of evaluating *UnaryExpression*.
2. If *val* is a [Reference Record](#), then
 - a. If `IsUnresolvableReference(val)` is **true**, return **"undefined"**.
3. Set *val* to ? `GetValue(val)`.
4. Return a String according to [Table 37](#).

Table 37: `typeof` Operator Results

Type of <i>val</i>	Result
Undefined	"undefined"
Null	"object"
Boolean	"boolean"
Number	"number"
String	"string"
Symbol	"symbol"
BigInt	"bigint"
Object (does not implement <code>[[Call]]</code>)	"object"
Object (implements <code>[[Call]]</code>)	"function"

NOTE An additional entry related to `[[IsHTMLDDA]]` Internal Slot can be found in [B.3.7.3](#).

13.5.4 Unary + Operator

13.5.4.1 Runtime Semantics: Evaluation

UnaryExpression : + *UnaryExpression*

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

13.5.5 Unary – Operator

13.5.5.1 Runtime Semantics: Evaluation

UnaryExpression : - *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ? **ToNumeric**(? **GetValue**(*expr*)).
3. Let *T* be **Type**(*oldValue*).
4. Return ! *T*::**unaryMinus**(*oldValue*).

13.5.6 Bitwise NOT Operator (~)

13.5.6.1 Runtime Semantics: Evaluation

UnaryExpression : ~ *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ? **ToNumeric**(? **GetValue**(*expr*)).
3. Let *T* be **Type**(*oldValue*).
4. Return ! *T*::**bitwiseNOT**(*oldValue*).

13.5.7 Logical NOT Operator (!)

13.5.7.1 Runtime Semantics: Evaluation

UnaryExpression : ! *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ! **ToBoolean**(? **GetValue**(*expr*)).
3. If *oldValue* is **true**, return **false**.
4. Return **true**.

13.6 Exponentiation Operator

Syntax

```
ExponentiationExpression [Yield, Await] :  
    UnaryExpression [?Yield, ?Await]  
    UpdateExpression [?Yield, ?Await] ** ExponentiationExpression [?Yield, ?Await]
```

13.6.1 Runtime Semantics: Evaluation

ExponentiationExpression : *UpdateExpression* ** *ExponentiationExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*UpdateExpression*, **, *ExponentiationExpression*).

13.7 Multiplicative Operators

Syntax

```
MultiplicativeExpression [Yield, Await] :  
    ExponentiationExpression [?Yield, ?Await]  
    MultiplicativeExpression [?Yield, ?Await] MultiplicativeOperator ExponentiationExpression [?Yield, ?Await]
```

MultiplicativeOperator : **one of**

* / %

NOTE

- The * operator performs multiplication, producing the product of its operands.
- The / operator performs division, producing the quotient of its operands.
- The % operator yields the remainder of its operands from an implied division.

13.7.1 Runtime Semantics: Evaluation

MultiplicativeExpression : *MultiplicativeExpression* *MultiplicativeOperator* *ExponentiationExpression*

1. Let *opText* be the source text matched by *MultiplicativeOperator*.
2. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*MultiplicativeExpression*, *opText*, *ExponentiationExpression*).

13.8 Additive Operators

Syntax

```
AdditiveExpression [Yield, Await] :  
    MultiplicativeExpression [?Yield, ?Await]  
    AdditiveExpression [?Yield, ?Await] + MultiplicativeExpression [?Yield, ?Await]  
    AdditiveExpression [?Yield, ?Await] - MultiplicativeExpression [?Yield, ?Await]
```

13.8.1 The Addition Operator (+)

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

13.8.1.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*AdditiveExpression*, +, *MultiplicativeExpression*).

13.8.2 The Subtraction Operator (-)

NOTE The - operator performs subtraction, producing the difference of its operands.

13.8.2.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* - *MultiplicativeExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*AdditiveExpression*, -, *MultiplicativeExpression*).

13.9 Bitwise Shift Operators

Syntax

```
ShiftExpression [Yield, Await] :  
  AdditiveExpression [?Yield, ?Await]  
  ShiftExpression [?Yield, ?Await] << AdditiveExpression [?Yield, ?Await]  
  ShiftExpression [?Yield, ?Await] >> AdditiveExpression [?Yield, ?Await]  
  ShiftExpression [?Yield, ?Await] >>> AdditiveExpression [?Yield, ?Await]
```

13.9.1 The Left Shift Operator (<<)

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

13.9.1.1 Runtime Semantics: Evaluation

ShiftExpression : *ShiftExpression* << *AdditiveExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*ShiftExpression*, <<, *AdditiveExpression*).

13.9.2 The Signed Right Shift Operator (>>)

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

13.9.2.1 Runtime Semantics: Evaluation

ShiftExpression : *ShiftExpression* >> *AdditiveExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*ShiftExpression*, >>, *AdditiveExpression*).

13.9.3 The Unsigned Right Shift Operator (>>>)

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

13.9.3.1 Runtime Semantics: Evaluation

ShiftExpression : *ShiftExpression* >>> *AdditiveExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*ShiftExpression*, >>>, *AdditiveExpression*).

13.10 Relational Operators

NOTE 1 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[In, Yield, Await] :  
  ShiftExpression[?Yield, ?Await]  
  RelationalExpression[?In, ?Yield, ?Await] < ShiftExpression[?Yield, ?Await]  
  RelationalExpression[?In, ?Yield, ?Await] > ShiftExpression[?Yield, ?Await]  
  RelationalExpression[?In, ?Yield, ?Await] <= ShiftExpression[?Yield, ?Await]  
  RelationalExpression[?In, ?Yield, ?Await] >= ShiftExpression[?Yield, ?Await]  
  RelationalExpression[?In, ?Yield, ?Await] instanceof ShiftExpression[?Yield, ?Await]  
  [+In] RelationalExpression[+In, ?Yield, ?Await] in ShiftExpression[?Yield, ?Await]
```

NOTE 2 The [In] grammar parameter is needed to avoid confusing the **in** operator in a relational expression with the **in** operator in a **for** statement.

13.10.1 Runtime Semantics: Evaluation

RelationalExpression : *RelationalExpression* < *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.

2. Let *lval* be ? `GetValue(lref)`.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Abstract Relational Comparison` *lval* < *rval*.
6. `ReturnIfAbrupt(r)`.
7. 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. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Abstract Relational Comparison` *rval* < *lval* with *LeftFirst* equal to **false**.
6. `ReturnIfAbrupt(r)`.
7. 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. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Abstract Relational Comparison` *rval* < *lval* with *LeftFirst* equal to **false**.
6. `ReturnIfAbrupt(r)`.
7. 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. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Abstract Relational Comparison` *lval* < *rval*.
6. `ReturnIfAbrupt(r)`.
7. 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. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Return ? `InstanceofOperator(lval, rval)`.

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

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be ? `GetValue(lref)`.
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be ? `GetValue(rref)`.

5. If `Type(rval)` is not `Object`, throw a `TypeError` exception.
6. Return `? HasProperty(rval, ? ToPropertyKey(lval))`.

13.10.2 InstanceofOperator (*V*, *target*)

The abstract operation `InstanceofOperator` takes arguments *V* (an [ECMAScript language value](#)) and *target* (an [ECMAScript language value](#)). It implements the generic algorithm for determining if *V* is an instance of *target* either by consulting *target*'s `@@hasInstance` method or, if absent, determining whether the value of *target*'s **"prototype"** property is present in *V*'s prototype chain. It performs the following steps when called:

1. If `Type(target)` is not `Object`, throw a `TypeError` exception.
2. Let *instOfHandler* be `? GetMethod(target, @@hasInstance)`.
3. If *instOfHandler* is not `undefined`, then
 - a. Return `! ToBoolean(? Call(instOfHandler, target, « V »))`.
4. If `IsCallable(target)` is `false`, throw a `TypeError` exception.
5. Return `? OrdinaryHasInstance(target, V)`.

NOTE Steps 4 and 5 provide compatibility with previous editions of ECMAScript that did not use a `@@hasInstance` method to define the **instanceof** operator semantics. If an object does not define or inherit `@@hasInstance` it uses the default **instanceof** semantics.

13.11 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 [In, Yield, Await] :
  RelationalExpression [?In, ?Yield, ?Await]
EqualityExpression [?In, ?Yield, ?Await] == RelationalExpression [?In, ?Yield, ?Await]
EqualityExpression [?In, ?Yield, ?Await] != RelationalExpression [?In, ?Yield, ?Await]
EqualityExpression [?In, ?Yield, ?Await] === RelationalExpression [?In, ?Yield, ?Await]
EqualityExpression [?In, ?Yield, ?Await] !== RelationalExpression [?In, ?Yield, ?Await]

```

13.11.1 Runtime Semantics: Evaluation

EqualityExpression : *EqualityExpression* == *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be `? GetValue(lref)`.
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be `? GetValue(rref)`.
5. 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. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Abstract Equality Comparison rval == lval`.
6. `ReturnIfAbrupt(r)`.
7. 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. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. 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. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be ? `GetValue(rref)`.
5. Let *r* be the result of performing `Strict Equality Comparison rval === lval`.
6. **Assert**: *r* is a normal completion.
7. If *r*.[[Value]] 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**.

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.

13.12 Binary Bitwise Operators

Syntax

BitwiseANDExpression [*In*, *Yield*, *Await*] :

EqualityExpression [*?In*, *?Yield*, *?Await*]

BitwiseANDExpression [*?In*, *?Yield*, *?Await*] & *EqualityExpression* [*?In*, *?Yield*, *?Await*]

BitwiseXORExpression [*In*, *Yield*, *Await*] :

BitwiseANDExpression [*?In*, *?Yield*, *?Await*]

BitwiseXORExpression [*?In*, *?Yield*, *?Await*] ^ *BitwiseANDExpression* [*?In*, *?Yield*, *?Await*]

BitwiseORExpression [*In*, *Yield*, *Await*] :

BitwiseXORExpression [*?In*, *?Yield*, *?Await*]

BitwiseORExpression [*?In*, *?Yield*, *?Await*] | *BitwiseXORExpression* [*?In*, *?Yield*, *?Await*]

13.12.1 Runtime Semantics: Evaluation

BitwiseANDExpression : *BitwiseANDExpression* & *EqualityExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*BitwiseANDExpression*, &, *EqualityExpression*).

BitwiseXORExpression : *BitwiseXORExpression* ^ *BitwiseANDExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*BitwiseXORExpression*, ^, *BitwiseANDExpression*).

BitwiseORExpression : *BitwiseORExpression* | *BitwiseXORExpression*

1. Return ? [EvaluateStringOrNumericBinaryExpression](#)(*BitwiseORExpression*, |, *BitwiseXORExpression*).

13.13 Binary Logical Operators

Syntax

LogicalANDExpression [*In*, *Yield*, *Await*] :

BitwiseORExpression [*?In*, *?Yield*, *?Await*]

LogicalANDExpression [*?In*, *?Yield*, *?Await*] && *BitwiseORExpression* [*?In*, *?Yield*, *?Await*]

LogicalORExpression [*In*, *Yield*, *Await*] :

LogicalANDExpression [*?In*, *?Yield*, *?Await*]

LogicalORExpression [*?In*, *?Yield*, *?Await*] || *LogicalANDExpression* [*?In*, *?Yield*, *?Await*]

CoalesceExpression [*In*, *Yield*, *Await*] :

CoalesceExpressionHead [*?In*, *?Yield*, *?Await*] ?? *BitwiseORExpression* [*?In*, *?Yield*, *?Await*]

CoalesceExpressionHead [*In*, *Yield*, *Await*] :

CoalesceExpression [*?In*, *?Yield*, *?Await*]

BitwiseORExpression [*?In*, *?Yield*, *?Await*]

ShortCircuitExpression [*In*, *Yield*, *Await*] :

LogicalORExpression [*?In*, *?Yield*, *?Await*]

CoalesceExpression [*?In*, *?Yield*, *?Await*]

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.

13.13.1 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. If *lbool* is **false**, return *lval*.
5. Let *rref* be the result of evaluating *BitwiseORExpression*.
6. Return ? *GetValue*(*rref*).

LogicalORExpression : *LogicalORExpression* || *LogicalANDExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be ? *GetValue*(*lref*).
3. Let *lbool* be ! *ToBoolean*(*lval*).
4. If *lbool* is **true**, return *lval*.
5. Let *rref* be the result of evaluating *LogicalANDExpression*.
6. Return ? *GetValue*(*rref*).

CoalesceExpression : *CoalesceExpressionHead* ?? *BitwiseORExpression*

1. Let *lref* be the result of evaluating *CoalesceExpressionHead*.
2. Let *lval* be ? *GetValue*(*lref*).
3. If *lval* is **undefined** or **null**, then
 - a. Let *rref* be the result of evaluating *BitwiseORExpression*.
 - b. Return ? *GetValue*(*rref*).
4. Otherwise, return *lval*.

13.14 Conditional Operator (? :)

Syntax

```
ConditionalExpression [In, Yield, Await] :  
  ShortCircuitExpression [?In, ?Yield, ?Await]  
  ShortCircuitExpression [?In, ?Yield, ?Await] ? AssignmentExpression [+In, ?Yield, ?Await] :  
    AssignmentExpression [?In, ?Yield, ?Await]
```

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.

13.14.1 Runtime Semantics: Evaluation

ConditionalExpression : *ShortCircuitExpression* ? *AssignmentExpression* : *AssignmentExpression*

1. Let *lref* be the result of evaluating *ShortCircuitExpression*.
2. Let *lval* be ! **ToBoolean**(? **GetValue**(*lref*)).
3. If *lval* is **true**, then
 - a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
 - b. Return ? **GetValue**(*trueRef*).
4. Else,
 - a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
 - b. Return ? **GetValue**(*falseRef*).

13.15 Assignment Operators

Syntax

```
AssignmentExpression [In, Yield, Await] :  
  ConditionalExpression [?In, ?Yield, ?Await]  
  [+Yield] YieldExpression [?In, ?Await]  
  ArrowFunction [?In, ?Yield, ?Await]  
  AsyncArrowFunction [?In, ?Yield, ?Await]  
  LeftHandSideExpression [?Yield, ?Await] = AssignmentExpression [?In, ?Yield, ?Await]  
  LeftHandSideExpression [?Yield, ?Await] AssignmentOperator AssignmentExpression [?In, ?Yield, ?Await]  
  LeftHandSideExpression [?Yield, ?Await] &&= AssignmentExpression [?In, ?Yield, ?Await]  
  LeftHandSideExpression [?Yield, ?Await] ||= AssignmentExpression [?In, ?Yield, ?Await]  
  LeftHandSideExpression [?Yield, ?Await] ??= AssignmentExpression [?In, ?Yield, ?Await]
```

AssignmentOperator : **one of**

***= /= %= += -= <<= >>= >>>= &= ^= |= **=**

13.15.1 Static Semantics: Early Errors

AssignmentExpression : *LeftHandSideExpression* = *AssignmentExpression*

If *LeftHandSideExpression* is an *ObjectLiteral* or an *ArrayLiteral*, the following Early Error rules are applied:

- It is a Syntax Error if *LeftHandSideExpression* is not **covering** an *AssignmentPattern*.
- All Early Error rules for *AssignmentPattern* and its derived productions also apply to the *AssignmentPattern* that is **covered** by *LeftHandSideExpression*.

If *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral*, the following Early Error rule is applied:

- It is a Syntax Error if **AssignmentTargetType** of *LeftHandSideExpression* is not **simple**.

AssignmentExpression :

LeftHandSideExpression *AssignmentOperator* *AssignmentExpression*

LeftHandSideExpression **&&=** *AssignmentExpression*

LeftHandSideExpression **||=** *AssignmentExpression*

LeftHandSideExpression **??=** *AssignmentExpression*

- It is a Syntax Error if **AssignmentTargetType** of *LeftHandSideExpression* is not **simple**.

13.15.2 Runtime Semantics: Evaluation

AssignmentExpression : *LeftHandSideExpression* = *AssignmentExpression*

1. If *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral*, then
 - a. Let *lref* be the result of evaluating *LeftHandSideExpression*.
 - b. **ReturnIfAbrupt**(*lref*).
 - c. If **IsAnonymousFunctionDefinition**(*AssignmentExpression*) and **IsIdentifierRef** of *LeftHandSideExpression* are both **true**, then
 - i. Let *rval* be **NamedEvaluation** of *AssignmentExpression* with argument *lref*.[[ReferencedName]].
 - d. Else,
 - i. Let *rref* be the result of evaluating *AssignmentExpression*.
 - ii. Let *rval* be ? **GetValue**(*rref*).
 - e. Perform ? **PutValue**(*lref*, *rval*).
 - f. Return *rval*.
2. Let *assignmentPattern* be the *AssignmentPattern* that is **covered** by *LeftHandSideExpression*.
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Perform ? **DestructuringAssignmentEvaluation** of *assignmentPattern* using *rval* as the argument.
6. Return *rval*.

AssignmentExpression : *LeftHandSideExpression* *AssignmentOperator* *AssignmentExpression*

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Let *rval* be ? **GetValue**(*rref*).
5. Let *assignmentOpText* be the source text matched by *AssignmentOperator*.
6. Let *opText* be the sequence of Unicode code points associated with *assignmentOpText* in the following table:

<i>assignmentOpText</i>	<i>opText</i>
**=	**
*=	*
/=	/
%=	%
+=	+
-=	-
<<=	<<
>>=	>>
>>>=	>>>
&=	&
^=	^
 =	

- Let r be `ApplyStringOrNumericBinaryOperator(lval, opText, rval)`.
- Perform ? `PutValue(lref, r)`.
- Return r .

AssignmentExpression : *LeftHandSideExpression* **&&=** *AssignmentExpression*

- Let $lref$ be the result of evaluating *LeftHandSideExpression*.
- Let $lval$ be ? `GetValue(lref)`.
- Let $lbool$ be ! `ToBoolean(lval)`.
- If $lbool$ is **false**, return $lval$.
- If `IsAnonymousFunctionDefinition(AssignmentExpression)` is **true** and `IsIdentifierRef` of *LeftHandSideExpression* is **true**, then
 - Let $rval$ be `NamedEvaluation` of *AssignmentExpression* with argument $lref.[[ReferencedName]]$.
- Else,
 - Let $rref$ be the result of evaluating *AssignmentExpression*.
 - Let $rval$ be ? `GetValue(rref)`.
- Perform ? `PutValue(lref, rval)`.
- Return $rval$.

AssignmentExpression : *LeftHandSideExpression* **||=** *AssignmentExpression*

- Let $lref$ be the result of evaluating *LeftHandSideExpression*.
- Let $lval$ be ? `GetValue(lref)`.
- Let $lbool$ be ! `ToBoolean(lval)`.
- If $lbool$ is **true**, return $lval$.
- If `IsAnonymousFunctionDefinition(AssignmentExpression)` is **true** and `IsIdentifierRef` of *LeftHandSideExpression* is **true**, then
 - Let $rval$ be `NamedEvaluation` of *AssignmentExpression* with argument $lref.[[ReferencedName]]$.
- Else,
 - Let $rref$ be the result of evaluating *AssignmentExpression*.
 - Let $rval$ be ? `GetValue(rref)`.
- Perform ? `PutValue(lref, rval)`.
- Return $rval$.

AssignmentExpression : *LeftHandSideExpression* **??=** *AssignmentExpression*

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *lval* be ? **GetValue**(*lref*).
3. If *lval* is neither **undefined** nor **null**, return *lval*.
4. If **IsAnonymousFunctionDefinition**(*AssignmentExpression*) is **true** and **IsIdentifierRef** of *LeftHandSideExpression* is **true**, then
 - a. Let *rval* be **NamedEvaluation** of *AssignmentExpression* with argument *lref*.[[ReferencedName]].
5. Else,
 - a. Let *rref* be the result of evaluating *AssignmentExpression*.
 - b. Let *rval* be ? **GetValue**(*rref*).
6. Perform ? **PutValue**(*lref*, *rval*).
7. Return *rval*.

NOTE

When this expression occurs within **strict mode code**, it is a runtime error if *lref* in step 1.e, 2, 2, 2, 2 is an unresolvable reference. If it is, a **ReferenceError** exception is thrown. Additionally, it is a runtime error if the *lref* in step 8, 7, 7, 6 is 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.

13.15.3 ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*)

The abstract operation **ApplyStringOrNumericBinaryOperator** takes arguments *lval* (an **ECMAScript language value**), *opText* (a sequence of Unicode code points), and *rval* (an **ECMAScript language value**). It performs the following steps when called:

1. **Assert**: *opText* is present in the table in step 8.
2. If *opText* is **+**, then
 - a. Let *lprim* be ? **ToPrimitive**(*lval*).
 - b. Let *rprim* be ? **ToPrimitive**(*rval*).
 - c. If **Type**(*lprim*) is **String** or **Type**(*rprim*) is **String**, then
 - i. Let *lstr* be ? **ToString**(*lprim*).
 - ii. Let *rstr* be ? **ToString**(*rprim*).
 - iii. Return the **string-concatenation** of *lstr* and *rstr*.
 - d. Set *lval* to *lprim*.
 - e. Set *rval* to *rprim*.
3. NOTE: At this point, it must be a numeric operation.
4. Let *lnum* be ? **ToNumeric**(*lval*).
5. Let *rnum* be ? **ToNumeric**(*rval*).
6. If **Type**(*lnum*) is different from **Type**(*rnum*), throw a **TypeError** exception.
7. Let *T* be **Type**(*lnum*).
8. Let *operation* be the abstract operation associated with *opText* in the following table:

<i>opText</i>	<i>operation</i>
**	<i>T</i> ::exponentiate
*	<i>T</i> ::multiply
/	<i>T</i> ::divide

%	<i>T::remainder</i>
+	<i>T::add</i>
-	<i>T::subtract</i>
<<	<i>T::leftShift</i>
>>	<i>T::signedRightShift</i>
>>>	<i>T::unsignedRightShift</i>
&	<i>T::bitwiseAND</i>
^	<i>T::bitwiseXOR</i>
 	<i>T::bitwiseOR</i>

9. Return ? *operation*(*lnum*, *rnum*).

NOTE 1 No hint is provided in the calls to [ToPrimitive](#) in steps 2.a and 2.b. All standard objects except Date objects handle the absence of a hint as if `number` were given; Date objects handle the absence of a hint as if `string` were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 2.c differs from step 3 of the [Abstract Relational Comparison](#) algorithm, by using the logical-or operation instead of the logical-and operation.

13.15.4 EvaluateStringOrNumericBinaryExpression (*leftOperand*, *opText*, *rightOperand*)

The abstract operation `EvaluateStringOrNumericBinaryExpression` takes arguments *leftOperand* (a [Parse Node](#)), *opText* (a sequence of Unicode code points), and *rightOperand* (a [Parse Node](#)). It performs the following steps when called:

1. Let *lref* be the result of evaluating *leftOperand*.
2. Let *lval* be ? [GetValue](#)(*lref*).
3. Let *rref* be the result of evaluating *rightOperand*.
4. Let *rval* be ? [GetValue](#)(*rref*).
5. Return ? [ApplyStringOrNumericBinaryOperator](#)(*lval*, *opText*, *rval*).

13.15.5 Destructuring Assignment

Supplemental Syntax

In certain circumstances when processing an instance of the production *AssignmentExpression* : *LeftHandSideExpression* = *AssignmentExpression* the interpretation of *LeftHandSideExpression* is refined using the following grammar:

```
AssignmentPattern [Yield, Await] :
    ObjectAssignmentPattern [?Yield, ?Await]
    ArrayAssignmentPattern [?Yield, ?Await]
```

```
ObjectAssignmentPattern [Yield, Await] :
    { }
```

```

{ AssignmentRestProperty [?Yield, ?Await] }
{ AssignmentPropertyList [?Yield, ?Await] }
{ AssignmentPropertyList [?Yield, ?Await] , AssignmentRestProperty [?Yield, ?Await] opt }

```

```

ArrayAssignmentPattern [Yield, Await] :
  [ Elisionopt AssignmentRestElement [?Yield, ?Await] opt ]
  [ AssignmentElementList [?Yield, ?Await] ]
  [ AssignmentElementList [?Yield, ?Await] , Elisionopt AssignmentRestElement [?Yield, ?Await] opt ]

```

```

AssignmentRestProperty [Yield, Await] :
  ... DestructuringAssignmentTarget [?Yield, ?Await]

```

```

AssignmentPropertyList [Yield, Await] :
  AssignmentProperty [?Yield, ?Await]
  AssignmentPropertyList [?Yield, ?Await] , AssignmentProperty [?Yield, ?Await]

```

```

AssignmentElementList [Yield, Await] :
  AssignmentElisionElement [?Yield, ?Await]
  AssignmentElementList [?Yield, ?Await] , AssignmentElisionElement [?Yield, ?Await]

```

```

AssignmentElisionElement [Yield, Await] :
  Elisionopt AssignmentElement [?Yield, ?Await]

```

```

AssignmentProperty [Yield, Await] :
  IdentifierReference [?Yield, ?Await] Initializer [+In, ?Yield, ?Await] opt
  PropertyName [?Yield, ?Await] : AssignmentElement [?Yield, ?Await]

```

```

AssignmentElement [Yield, Await] :
  DestructuringAssignmentTarget [?Yield, ?Await] Initializer [+In, ?Yield, ?Await] opt

```

```

AssignmentRestElement [Yield, Await] :
  ... DestructuringAssignmentTarget [?Yield, ?Await]

```

```

DestructuringAssignmentTarget [Yield, Await] :
  LeftHandSideExpression [?Yield, ?Await]

```

13.15.5.1 Static Semantics: Early Errors

AssignmentProperty : *IdentifierReference* *Initializer*_{opt}

- It is a Syntax Error if *AssignmentTargetType* of *IdentifierReference* is not **simple**.

AssignmentRestProperty : ... *DestructuringAssignmentTarget*

- It is a Syntax Error if *DestructuringAssignmentTarget* is an *ArrayLiteral* or an *ObjectLiteral*.

DestructuringAssignmentTarget : *LeftHandSideExpression*

If *LeftHandSideExpression* is an *ObjectLiteral* or an *ArrayLiteral*, the following Early Error rules are applied:

- It is a Syntax Error if *LeftHandSideExpression* is not **covering** an *AssignmentPattern*.
- All Early Error rules for *AssignmentPattern* and its derived productions also apply to the *AssignmentPattern* that is **covered** by *LeftHandSideExpression*.

If *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral*, the following Early Error rule is applied:

- It is a Syntax Error if **AssignmentTargetType** of *LeftHandSideExpression* is not **simple**.

13.15.5.2 Runtime Semantics: DestructuringAssignmentEvaluation

With parameter *value*.

ObjectAssignmentPattern : { }

1. Perform ? **RequireObjectCoercible**(*value*).
2. Return **NormalCompletion**(empty).

ObjectAssignmentPattern :

{ *AssignmentPropertyList* }
{ *AssignmentPropertyList* , }

1. Perform ? **RequireObjectCoercible**(*value*).
2. Perform ? **PropertyDestructuringAssignmentEvaluation** for *AssignmentPropertyList* using *value* as the argument.
3. Return **NormalCompletion**(empty).

ArrayAssignmentPattern : []

1. Let *iteratorRecord* be ? **GetIterator**(*value*).
2. Return ? **IteratorClose**(*iteratorRecord*, **NormalCompletion**(empty)).

ArrayAssignmentPattern : [*Elision*]

1. Let *iteratorRecord* be ? **GetIterator**(*value*).
2. Let *result* be **IteratorDestructuringAssignmentEvaluation** of *Elision* with argument *iteratorRecord*.
3. If *iteratorRecord*.[[Done]] is **false**, return ? **IteratorClose**(*iteratorRecord*, *result*).
4. Return *result*.

ArrayAssignmentPattern : [*Elision*_{opt} *AssignmentRestElement*]

1. Let *iteratorRecord* be ? **GetIterator**(*value*).
2. If *Elision* is present, then
 - a. Let *status* be **IteratorDestructuringAssignmentEvaluation** of *Elision* with argument *iteratorRecord*.
 - b. If *status* is an **abrupt completion**, then
 - i. **Assert**: *iteratorRecord*.[[Done]] is **true**.
 - ii. Return **Completion**(*status*).
3. Let *result* be **IteratorDestructuringAssignmentEvaluation** of *AssignmentRestElement* with argument *iteratorRecord*.
4. If *iteratorRecord*.[[Done]] is **false**, return ? **IteratorClose**(*iteratorRecord*, *result*).
5. Return *result*.

ArrayAssignmentPattern : [*AssignmentElementList*]

1. Let *iteratorRecord* be ? [GetIterator](#)(*value*).
2. Let *result* be [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElementList* with argument *iteratorRecord*.
3. If *iteratorRecord*.[[Done]] is **false**, return ? [IteratorClose](#)(*iteratorRecord*, *result*).
4. Return *result*.

ArrayAssignmentPattern : [*AssignmentElementList* , *Elision*_{opt} *AssignmentRestElement*_{opt}]

1. Let *iteratorRecord* be ? [GetIterator](#)(*value*).
2. Let *status* be [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElementList* with argument *iteratorRecord*.
3. If *status* is an **abrupt completion**, then
 - a. If *iteratorRecord*.[[Done]] is **false**, return ? [IteratorClose](#)(*iteratorRecord*, *status*).
 - b. Return [Completion](#)(*status*).
4. If *Elision* is present, then
 - a. Set *status* to the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *Elision* with *iteratorRecord* as the argument.
 - b. If *status* is an **abrupt completion**, then
 - i. **Assert**: *iteratorRecord*.[[Done]] is **true**.
 - ii. Return [Completion](#)(*status*).
5. If *AssignmentRestElement* is present, then
 - a. Set *status* to the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentRestElement* with *iteratorRecord* as the argument.
6. If *iteratorRecord*.[[Done]] is **false**, return ? [IteratorClose](#)(*iteratorRecord*, *status*).
7. Return [Completion](#)(*status*).

ObjectAssignmentPattern : { *AssignmentRestProperty* }

1. Perform ? [RequireObjectCoercible](#)(*value*).
2. Let *excludedNames* be a new empty [List](#).
3. Return the result of performing [RestDestructuringAssignmentEvaluation](#) of *AssignmentRestProperty* with *value* and *excludedNames* as the arguments.

ObjectAssignmentPattern : { *AssignmentPropertyList* , *AssignmentRestProperty* }

1. Perform ? [RequireObjectCoercible](#)(*value*).
2. Let *excludedNames* be ? [PropertyDestructuringAssignmentEvaluation](#) of *AssignmentPropertyList* with argument *value*.
3. Return the result of performing [RestDestructuringAssignmentEvaluation](#) of *AssignmentRestProperty* with arguments *value* and *excludedNames*.

13.15.5.3 Runtime Semantics: PropertyDestructuringAssignmentEvaluation

With parameter *value*.

NOTE The following operations collect a list of all destructured property names.

AssignmentPropertyList : *AssignmentPropertyList* , *AssignmentProperty*

1. Let *propertyName* be ? [PropertyDestructuringAssignmentEvaluation](#) of *AssignmentPropertyList* with argument *value*.
2. Let *nextNames* be ? [PropertyDestructuringAssignmentEvaluation](#) of *AssignmentProperty* with argument *value*.
3. Append each item in *nextNames* to the end of *propertyName*.
4. Return *propertyName*.

AssignmentProperty : *IdentifierReference* *Initializer*_{opt}

1. Let *P* be [StringValue](#) of *IdentifierReference*.
2. Let *lref* be ? [ResolveBinding](#)(*P*).
3. Let *v* be ? [GetV](#)(*value*, *P*).
4. If *Initializer*_{opt} is present and *v* is **undefined**, then
 - a. If [IsAnonymousFunctionDefinition](#)(*Initializer*) is **true**, then
 - i. Set *v* to the result of performing [NamedEvaluation](#) for *Initializer* with argument *P*.
 - b. Else,
 - i. Let *defaultValue* be the result of evaluating *Initializer*.
 - ii. Set *v* to ? [GetValue](#)(*defaultValue*).
5. Perform ? [PutValue](#)(*lref*, *v*).
6. Return a [List](#) whose sole element is *P*.

AssignmentProperty : *PropertyName* : *AssignmentElement*

1. Let *name* be the result of evaluating *PropertyName*.
2. [ReturnIfAbrupt](#)(*name*).
3. Perform ? [KeyedDestructuringAssignmentEvaluation](#) of *AssignmentElement* with *value* and *name* as the arguments.
4. Return a [List](#) whose sole element is *name*.

13.15.5.4 Runtime Semantics: RestDestructuringAssignmentEvaluation

With parameters *value* and *excludedNames*.

AssignmentRestProperty : ... *DestructuringAssignmentTarget*

1. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
2. [ReturnIfAbrupt](#)(*lref*).
3. Let *restObj* be ! [OrdinaryObjectCreate](#)(%*Object.prototype*%).
4. Perform ? [CopyDataProperties](#)(*restObj*, *value*, *excludedNames*).
5. Return [PutValue](#)(*lref*, *restObj*).

13.15.5.5 Runtime Semantics: IteratorDestructuringAssignmentEvaluation

With parameter *iteratorRecord*.

AssignmentElementList : *AssignmentElisionElement*

1. Return the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElisionElement* using *iteratorRecord* as the argument.

AssignmentElementList : *AssignmentElementList* , *AssignmentElisionElement*

1. Perform ? [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElementList* using *iteratorRecord* as the argument.
2. Return the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElisionElement* using *iteratorRecord* as the argument.

AssignmentElisionElement : *AssignmentElement*

1. Return the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElement* with *iteratorRecord* as the argument.

AssignmentElisionElement : *Elision* *AssignmentElement*

1. Perform ? [IteratorDestructuringAssignmentEvaluation](#) of *Elision* with *iteratorRecord* as the argument.
2. Return the result of performing [IteratorDestructuringAssignmentEvaluation](#) of *AssignmentElement* with *iteratorRecord* as the argument.

Elision : ,

1. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
 - c. [ReturnIfAbrupt](#)(*next*).
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
2. Return [NormalCompletion](#)(empty).

Elision : *Elision* ,

1. Perform ? [IteratorDestructuringAssignmentEvaluation](#) of *Elision* with *iteratorRecord* as the argument.
2. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
 - c. [ReturnIfAbrupt](#)(*next*).
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
3. Return [NormalCompletion](#)(empty).

AssignmentElement : *DestructuringAssignmentTarget* *Initializer*_{opt}

1. If *DestructuringAssignmentTarget* is neither an *ObjectLiteral* nor an *ArrayLiteral*, then
 - a. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
 - b. [ReturnIfAbrupt](#)(*lref*).
2. If *iteratorRecord*.[[Done]] is **false**, then
 - a. Let *next* be [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
 - c. [ReturnIfAbrupt](#)(*next*).
 - d. If *next* is **false**, set *iteratorRecord*.[[Done]] to **true**.
 - e. Else,
 - i. Let *value* be [IteratorValue](#)(*next*).
 - ii. If *value* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
 - iii. [ReturnIfAbrupt](#)(*value*).
3. If *iteratorRecord*.[[Done]] is **true**, let *value* be **undefined**.
4. If *Initializer* is present and *value* is **undefined**, then

- a. If `IsAnonymousFunctionDefinition(Initializer)` is **true** and `IsIdentifierRef` of `DestructuringAssignmentTarget` is **true**, then
 - i. Let v be ? `NamedEvaluation` of `Initializer` with argument `lref.[[ReferencedName]]`.
 - b. Else,
 - i. Let `defaultValue` be the result of evaluating `Initializer`.
 - ii. Let v be ? `GetValue(defaultValue)`.
5. Else, let v be `value`.
 6. If `DestructuringAssignmentTarget` is an `ObjectLiteral` or an `ArrayLiteral`, then
 - a. Let `nestedAssignmentPattern` be the `AssignmentPattern` that is **covered** by `DestructuringAssignmentTarget`.
 - b. Return the result of performing `DestructuringAssignmentEvaluation` of `nestedAssignmentPattern` with v as the argument.
 7. Return ? `PutValue(lref, v)`.

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 : ... *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 be 0.
4. Repeat, while `iteratorRecord.[[Done]]` is **false**,
 - a. Let `next` be `IteratorStep(iteratorRecord)`.
 - b. If `next` is an **abrupt completion**, set `iteratorRecord.[[Done]]` to **true**.
 - c. `ReturnIfAbrupt(next)`.
 - d. If `next` is **false**, set `iteratorRecord.[[Done]]` to **true**.
 - e. Else,
 - i. Let `nextValue` be `IteratorValue(next)`.
 - ii. If `nextValue` is an **abrupt completion**, set `iteratorRecord.[[Done]]` to **true**.
 - iii. `ReturnIfAbrupt(nextValue)`.
 - iv. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(n)), nextValue)`.
 - v. Set n to $n + 1$.
5. If `DestructuringAssignmentTarget` is neither an `ObjectLiteral` nor an `ArrayLiteral`, then
 - a. Return ? `PutValue(lref, A)`.
6. Let `nestedAssignmentPattern` be the `AssignmentPattern` that is **covered** by `DestructuringAssignmentTarget`.
7. Return the result of performing `DestructuringAssignmentEvaluation` of `nestedAssignmentPattern` with A as the argument.

13.15.5.6 Runtime Semantics: KeyedDestructuringAssignmentEvaluation

With parameters `value` and `propertyName`.

AssignmentElement : *DestructuringAssignmentTarget* *Initializer*_{opt}

1. If `DestructuringAssignmentTarget` is neither an `ObjectLiteral` nor an `ArrayLiteral`, then
 - a. Let `lref` be the result of evaluating `DestructuringAssignmentTarget`.

- b. `ReturnIfAbrupt(lref)`.
2. Let v be ? `GetV(value, propertyName)`.
3. If `Initializer` is present and v is **undefined**, then
 - a. If `IsAnonymousFunctionDefinition(Initializer)` and `IsIdentifierRef` of `DestructuringAssignmentTarget` are both **true**, then
 - i. Let $rhsValue$ be ? `NamedEvaluation` of `Initializer` with argument $lref.[[ReferencedName]]$.
 - b. Else,
 - i. Let $defaultValue$ be the result of evaluating `Initializer`.
 - ii. Let $rhsValue$ be ? `GetValue(defaultValue)`.
4. Else, let $rhsValue$ be v .
5. If `DestructuringAssignmentTarget` is an `ObjectLiteral` or an `ArrayLiteral`, then
 - a. Let $assignmentPattern$ be the `AssignmentPattern` that is **covered** by `DestructuringAssignmentTarget`.
 - b. Return the result of performing `DestructuringAssignmentEvaluation` of $assignmentPattern$ with $rhsValue$ as the argument.
6. Return ? `PutValue(lref, rhsValue)`.

13.16 Comma Operator (,)

Syntax

```

Expression [In, Yield, Await] :
  AssignmentExpression [?In, ?Yield, ?Await]
  Expression [?In, ?Yield, ?Await] , AssignmentExpression [?In, ?Yield, ?Await]

```

13.16.1 Runtime Semantics: Evaluation

Expression : *Expression* , *AssignmentExpression*

1. Let $lref$ be the result of evaluating *Expression*.
2. Perform ? `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.

14 ECMAScript Language: Statements and Declarations

Syntax

```

Statement [Yield, Await, Return] :
  BlockStatement [?Yield, ?Await, ?Return]
  VariableStatement [?Yield, ?Await]

```

EmptyStatement
ExpressionStatement [*?Yield*, *?Await*]
IfStatement [*?Yield*, *?Await*, *?Return*]
BreakableStatement [*?Yield*, *?Await*, *?Return*]
ContinueStatement [*?Yield*, *?Await*]
BreakStatement [*?Yield*, *?Await*]
[+Return] ReturnStatement [*?Yield*, *?Await*]
WithStatement [*?Yield*, *?Await*, *?Return*]
LabelledStatement [*?Yield*, *?Await*, *?Return*]
ThrowStatement [*?Yield*, *?Await*]
TryStatement [*?Yield*, *?Await*, *?Return*]
DebuggerStatement

Declaration [*Yield*, *Await*] :

- HoistableDeclaration* [*?Yield*, *?Await*, *~Default*]
- ClassDeclaration* [*?Yield*, *?Await*, *~Default*]
- LexicalDeclaration* [*+In*, *?Yield*, *?Await*]

HoistableDeclaration [*Yield*, *Await*, *Default*] :

- FunctionDeclaration* [*?Yield*, *?Await*, *?Default*]
- GeneratorDeclaration* [*?Yield*, *?Await*, *?Default*]
- AsyncFunctionDeclaration* [*?Yield*, *?Await*, *?Default*]
- AsyncGeneratorDeclaration* [*?Yield*, *?Await*, *?Default*]

BreakableStatement [*Yield*, *Await*, *Return*] :

- IterationStatement* [*?Yield*, *?Await*, *?Return*]
- SwitchStatement* [*?Yield*, *?Await*, *?Return*]

14.1 Statement Semantics

14.1.1 Runtime Semantics: Evaluation

HoistableDeclaration :

- GeneratorDeclaration*
- AsyncFunctionDeclaration*
- AsyncGeneratorDeclaration*

1. Return **NormalCompletion**(empty).

HoistableDeclaration : *FunctionDeclaration*

1. Return the result of evaluating *FunctionDeclaration*.

BreakableStatement :

IterationStatement

SwitchStatement

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

14.2 Block

Syntax

*BlockStatement*_[Yield, Await, Return] :

*Block*_[?Yield, ?Await, ?Return]

*Block*_[Yield, Await, Return] :

{ *StatementList*_[?Yield, ?Await, ?Return] *opt* }

*StatementList*_[Yield, Await, Return] :

*StatementListItem*_[?Yield, ?Await, ?Return]

*StatementList*_[?Yield, ?Await, ?Return] *StatementListItem*_[?Yield, ?Await, ?Return]

*StatementListItem*_[Yield, Await, Return] :

*Statement*_[?Yield, ?Await, ?Return]

*Declaration*_[?Yield, ?Await]

14.2.1 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*.

14.2.2 Runtime Semantics: Evaluation

Block : { }

1. Return *NormalCompletion*(empty).

Block : { *StatementList* }

1. Let *oldEnv* be the *running execution context*'s *LexicalEnvironment*.
2. Let *blockEnv* be *NewDeclarativeEnvironment*(*oldEnv*).
3. Perform *BlockDeclarationInstantiation*(*StatementList*, *blockEnv*).
4. Set the *running execution context*'s *LexicalEnvironment* to *blockEnv*.
5. Let *blockValue* be the result of evaluating *StatementList*.
6. Set the *running execution context*'s *LexicalEnvironment* to *oldEnv*.
7. Return *blockValue*.

NOTE 1 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. Return **Completion**(**UpdateEmpty**(*s*, *sl*)).

NOTE 2 The value of a *StatementList* is the value of the last value-producing item in the *StatementList*. For example, the following calls to the **eval** function all return the value 1:

```
eval("1;;;;;")
eval("1;{}")
eval("1;var a;")
```

14.2.3 BlockDeclarationInstantiation (*code*, *env*)

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

The abstract operation **BlockDeclarationInstantiation** takes arguments *code* (a **Parse Node**) and *env* (an **Environment Record**). *code* is the **Parse Node** corresponding to the body of the block. *env* is the **Environment Record** in which bindings are to be created. It performs the following steps when called:

1. **Assert**: *env* is a **declarative Environment Record**.
2. Let *declarations* be the **LexicallyScopedDeclarations** of *code*.
3. For each element *d* of *declarations*, do
 - a. For each element *dn* of the **BoundNames** of *d*, do
 - i. If **IsConstantDeclaration** of *d* is **true**, then
 1. Perform ! *env*.**CreateImmutableBinding**(*dn*, **true**).
 - ii. Else,
 1. Perform ! *env*.**CreateMutableBinding**(*dn*, **false**). NOTE: This step is replaced in section B.3.3.6.
 - b. If *d* is a *FunctionDeclaration*, a *GeneratorDeclaration*, an *AsyncFunctionDeclaration*, or an *AsyncGeneratorDeclaration*, then
 - i. Let *fn* be the sole element of the **BoundNames** of *d*.
 - ii. Let *fo* be **InstantiateFunctionObject** of *d* with argument *env*.
 - iii. Perform *env*.**InitializeBinding**(*fn*, *fo*). NOTE: This step is replaced in section B.3.3.6.

14.3 Declarations and the Variable Statement

14.3.1 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 [Environment Record](#) 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, Await] :  
  LetOrConst BindingList[?In, ?Yield, ?Await] ;
```

```
LetOrConst :  
  let  
  const
```

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

```
LexicalBinding[In, Yield, Await] :  
  BindingIdentifier[?Yield, ?Await] Initializer[?In, ?Yield, ?Await] opt  
  BindingPattern[?Yield, ?Await] Initializer[?In, ?Yield, ?Await]
```

14.3.1.1 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* *Initializer*_{opt}

- It is a Syntax Error if *Initializer* is not present and [IsConstantDeclaration](#) of the *LexicalDeclaration* containing this *LexicalBinding* is **true**.

14.3.1.2 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 `lhs` be `ResolveBinding(StringValue of BindingIdentifier)`.
2. Return `InitializeReferencedBinding(lhs, undefined)`.

NOTE A **static semantics** rule ensures that this form of `LexicalBinding` never occurs in a **const** declaration.

`LexicalBinding` : `BindingIdentifier Initializer`

1. Let `bindingId` be `StringValue` of `BindingIdentifier`.
2. Let `lhs` be `ResolveBinding(bindingId)`.
3. If `IsAnonymousFunctionDefinition(Initializer)` is **true**, then
 - a. Let `value` be `NamedEvaluation` of `Initializer` with argument `bindingId`.
4. Else,
 - a. Let `rhs` be the result of evaluating `Initializer`.
 - b. Let `value` be ? `GetValue(rhs)`.
5. Return `InitializeReferencedBinding(lhs, value)`.

`LexicalBinding` : `BindingPattern Initializer`

1. Let `rhs` be the result of evaluating `Initializer`.
2. Let `value` be ? `GetValue(rhs)`.
3. Let `env` be the **running execution context**'s `LexicalEnvironment`.
4. Return the result of performing `BindingInitialization` for `BindingPattern` using `value` and `env` as the arguments.

14.3.2 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 `Environment Record` 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 collectively 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

```
VariableStatement[Yield, Await] :
    var VariableDeclarationList[+In, ?Yield, ?Await] ;
```

```
VariableDeclarationList[In, Yield, Await] :
    VariableDeclaration[?In, ?Yield, ?Await]
    VariableDeclarationList[?In, ?Yield, ?Await] , VariableDeclaration[?In, ?Yield, ?Await]
```

VariableDeclaration [*In*, *Yield*, *Await*] :

BindingIdentifier [*?Yield*, *?Await*] *Initializer* [*?In*, *?Yield*, *?Await*] *opt*

BindingPattern [*?Yield*, *?Await*] *Initializer* [*?In*, *?Yield*, *?Await*]

14.3.2.1 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 *bindingId* be **StringValue** of *BindingIdentifier*.
2. Let *lhs* be ? **ResolveBinding**(*bindingId*).
3. If **IsAnonymousFunctionDefinition**(*Initializer*) is **true**, then
 - a. Let *value* be **NamedEvaluation** of *Initializer* with argument *bindingId*.
4. Else,
 - a. Let *rhs* be the result of evaluating *Initializer*.
 - b. Let *value* be ? **GetValue**(*rhs*).
5. Return ? **PutValue**(*lhs*, *value*).

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 ? **GetValue**(*rhs*).
3. Return the result of performing **BindingInitialization** for *BindingPattern* passing *rval* and **undefined** as arguments.

14.3.3 Destructuring Binding Patterns

Syntax

BindingPattern [*Yield*, *Await*] :

`ObjectBindingPattern` `[?Yield, ?Await]`

`ArrayBindingPattern` `[?Yield, ?Await]`

`ObjectBindingPattern` `[Yield, Await]` :

{ }

{ `BindingRestProperty` `[?Yield, ?Await]` }

{ `BindingPropertyList` `[?Yield, ?Await]` }

{ `BindingPropertyList` `[?Yield, ?Await]` , `BindingRestProperty` `[?Yield, ?Await]` `opt` }

`ArrayBindingPattern` `[Yield, Await]` :

[`Elision` `opt` `BindingRestElement` `[?Yield, ?Await]` `opt`]

[`BindingElementList` `[?Yield, ?Await]`]

[`BindingElementList` `[?Yield, ?Await]` , `Elision` `opt` `BindingRestElement` `[?Yield, ?Await]` `opt`]

`BindingRestProperty` `[Yield, Await]` :

... `BindingIdentifier` `[?Yield, ?Await]`

`BindingPropertyList` `[Yield, Await]` :

`BindingProperty` `[?Yield, ?Await]`

`BindingPropertyList` `[?Yield, ?Await]` , `BindingProperty` `[?Yield, ?Await]`

`BindingElementList` `[Yield, Await]` :

`BindingElisionElement` `[?Yield, ?Await]`

`BindingElementList` `[?Yield, ?Await]` , `BindingElisionElement` `[?Yield, ?Await]`

`BindingElisionElement` `[Yield, Await]` :

`Elision` `opt` `BindingElement` `[?Yield, ?Await]`

`BindingProperty` `[Yield, Await]` :

`SingleNameBinding` `[?Yield, ?Await]`

`PropertyName` `[?Yield, ?Await]` : `BindingElement` `[?Yield, ?Await]`

`BindingElement` `[Yield, Await]` :

`SingleNameBinding` `[?Yield, ?Await]`

`BindingPattern` `[?Yield, ?Await]` `Initializer` `[+In, ?Yield, ?Await]` `opt`

`SingleNameBinding` `[Yield, Await]` :

`BindingIdentifier` `[?Yield, ?Await]` `Initializer` `[+In, ?Yield, ?Await]` `opt`

`BindingRestElement` `[Yield, Await]` :

... `BindingIdentifier` `[?Yield, ?Await]`

... `BindingPattern` `[?Yield, ?Await]`

14.3.3.1 Runtime Semantics: PropertyBindingInitialization

With parameters *value* and *environment*.

NOTE These collect a list of all bound property names rather than just empty completion.

BindingPropertyList : *BindingPropertyList* , *BindingProperty*

1. Let *boundNames* be ? [PropertyBindingInitialization](#) of *BindingPropertyList* with arguments *value* and *environment*.
2. Let *nextNames* be ? [PropertyBindingInitialization](#) of *BindingProperty* with arguments *value* and *environment*.
3. Append each item in *nextNames* to the end of *boundNames*.
4. Return *boundNames*.

BindingProperty : *SingleNameBinding*

1. Let *name* be the string that is the only element of [BoundNames](#) of *SingleNameBinding*.
2. Perform ? [KeyedBindingInitialization](#) for *SingleNameBinding* using *value*, *environment*, and *name* as the arguments.
3. Return a [List](#) whose sole element is *name*.

BindingProperty : *PropertyName* : *BindingElement*

1. Let *P* be the result of evaluating *PropertyName*.
2. [ReturnIfAbrupt](#)(*P*).
3. Perform ? [KeyedBindingInitialization](#) of *BindingElement* with *value*, *environment*, and *P* as the arguments.
4. Return a [List](#) whose sole element is *P*.

14.3.3.2 Runtime Semantics: RestBindingInitialization

With parameters *value*, *environment*, and *excludedNames*.

BindingRestProperty : ... *BindingIdentifier*

1. Let *lhs* be ? [ResolveBinding](#)([StringValue](#) of *BindingIdentifier*, *environment*).
2. Let *restObj* be ! [OrdinaryObjectCreate](#)(%Object.prototype%).
3. Perform ? [CopyDataProperties](#)(*restObj*, *value*, *excludedNames*).
4. If *environment* is **undefined**, return [PutValue](#)(*lhs*, *restObj*).
5. Return [InitializeReferencedBinding](#)(*lhs*, *restObj*).

14.3.3.3 Runtime Semantics: KeyedBindingInitialization

With parameters *value*, *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* *Initializer*_{opt}

1. Let *v* be ? [GetV](#)(*value*, *propertyName*).
2. If *Initializer* is present and *v* is **undefined**, then

- a. Let *defaultValue* be the result of evaluating *Initializer*.
 - b. Set *v* to ? *GetValue*(*defaultValue*).
3. Return the result of performing *BindingInitialization* for *BindingPattern* passing *v* and *environment* as arguments.

SingleNameBinding : *BindingIdentifier* *Initializer*_{opt}

1. Let *bindingId* be *StringValue* of *BindingIdentifier*.
2. Let *lhs* be ? *ResolveBinding*(*bindingId*, *environment*).
3. Let *v* be ? *GetV*(*value*, *propertyName*).
4. If *Initializer* is present and *v* is **undefined**, then
 - a. If *IsAnonymousFunctionDefinition*(*Initializer*) is **true**, then
 - i. Set *v* to the result of performing *NamedEvaluation* for *Initializer* with argument *bindingId*.
 - b. Else,
 - i. Let *defaultValue* be the result of evaluating *Initializer*.
 - ii. Set *v* to ? *GetValue*(*defaultValue*).
5. If *environment* is **undefined**, return ? *PutValue*(*lhs*, *v*).
6. Return *InitializeReferencedBinding*(*lhs*, *v*).

14.4 Empty Statement

Syntax

EmptyStatement :
;

14.4.1 Runtime Semantics: Evaluation

EmptyStatement : ;

1. Return *NormalCompletion*(empty).

14.5 Expression Statement

Syntax

*ExpressionStatement*_[Yield, Await] :
[lookahead ∉ { {, function, async [no LineTerminator here] function, class, let [] } }
*Expression*_[+In, ?Yield, ?Await] ;

NOTE

An *ExpressionStatement* cannot start with a U+007B (LEFT CURLY BRACKET) because that might make it ambiguous with a *Block*. 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 **async function** because that would make it ambiguous with an *AsyncFunctionDeclaration* or a *AsyncGeneratorDeclaration*. 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*.

14.5.1 Runtime Semantics: Evaluation

ExpressionStatement : *Expression* ;

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return ? *GetValue*(*exprRef*).

14.6 The if Statement

Syntax

```
IfStatement[Yield, Await, Return] :  
  if ( Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return] else  
    Statement[?Yield, ?Await, ?Return]  
  if ( Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return] [lookahead ≠ else]
```

NOTE

The lookahead-restriction [lookahead ≠ **else**] resolves the classic "dangling else" problem in the usual way. That is, when the choice of associated **if** is otherwise ambiguous, the **else** is associated with the nearest (innermost) of the candidate **ifs**

14.6.1 Static Semantics: Early Errors

IfStatement :

```
if ( Expression ) Statement else Statement  
if ( Expression ) Statement
```

- It is a Syntax Error if *IsLabelledFunction*(*Statement*) is **true**.

NOTE

It is only necessary to apply this rule if the extension specified in [B.3.2](#) is implemented.

14.6.2 Runtime Semantics: Evaluation

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ! *ToBoolean*(? *GetValue*(*exprRef*)).
3. If *exprValue* is **true**, then

- a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
4. Else,
 - a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
5. Return `Completion(UpdateEmpty(stmtCompletion, undefined))`.

IfStatement : `if (Expression) Statement`

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be `! ToBoolean(? GetValue(exprRef))`.
3. If *exprValue* is **false**, then
 - a. Return `NormalCompletion(undefined)`.
4. Else,
 - a. Let *stmtCompletion* be the result of evaluating *Statement*.
 - b. Return `Completion(UpdateEmpty(stmtCompletion, undefined))`.

14.7 Iteration Statements

Syntax

```
IterationStatement [Yield, Await, Return] :
  DoWhileStatement [ ?Yield, ?Await, ?Return ]
  WhileStatement [ ?Yield, ?Await, ?Return ]
  ForStatement [ ?Yield, ?Await, ?Return ]
  ForInOfStatement [ ?Yield, ?Await, ?Return ]
```

14.7.1 Semantics

14.7.1.1 LoopContinues (*completion*, *labelSet*)

The abstract operation LoopContinues takes arguments *completion* and *labelSet*. It performs the following steps when called:

1. If *completion*.[[Type]] is normal, return **true**.
2. If *completion*.[[Type]] is not **continue**, return **false**.
3. If *completion*.[[Target]] is empty, return **true**.
4. If *completion*.[[Target]] is an element of *labelSet*, return **true**.
5. Return **false**.

NOTE Within the *Statement* part of an *IterationStatement* a *ContinueStatement* may be used to begin a new iteration.

14.7.1.2 Runtime Semantics: LoopEvaluation

With parameter *labelSet*.

IterationStatement : *DoWhileStatement*

1. Return ? [DoWhileLoopEvaluation](#) of *DoWhileStatement* with argument *labelSet*.

IterationStatement : *WhileStatement*

1. Return ? [WhileLoopEvaluation](#) of *WhileStatement* with argument *labelSet*.

IterationStatement : *ForStatement*

1. Return ? [ForLoopEvaluation](#) of *ForStatement* with argument *labelSet*.

IterationStatement : *ForInOfStatement*

1. Return ? [ForInOfLoopEvaluation](#) of *ForInOfStatement* with argument *labelSet*.

14.7.2 The **do-while** Statement

Syntax

```
DoWhileStatement [Yield, Await, Return] :
    do Statement [?Yield, ?Await, ?Return] while ( Expression [+In, ?Yield, ?Await] ) ;
```

14.7.2.1 Static Semantics: Early Errors

DoWhileStatement : do *Statement* while (*Expression*) ;

- It is a Syntax Error if [IsLabelledFunction](#)(*Statement*) is **true**.

NOTE It is only necessary to apply this rule if the extension specified in [B.3.2](#) is implemented.

14.7.2.2 Runtime Semantics: DoWhileLoopEvaluation

With parameter *labelSet*.

DoWhileStatement : do *Statement* while (*Expression*) ;

1. Let *V* be **undefined**.
2. Repeat,
 - a. Let *stmtResult* be the result of evaluating *Statement*.
 - b. If [LoopContinues](#)(*stmtResult*, *labelSet*) is **false**, return [Completion](#)([UpdateEmpty](#)(*stmtResult*, *V*)).
 - c. If *stmtResult*.[[Value]] is not empty, set *V* to *stmtResult*.[[Value]].
 - d. Let *exprRef* be the result of evaluating *Expression*.
 - e. Let *exprValue* be ? [GetValue](#)(*exprRef*).
 - f. If ! [ToBoolean](#)(*exprValue*) is **false**, return [NormalCompletion](#)(*V*).

14.7.3 The **while** Statement

Syntax

```
WhileStatement [Yield, Await, Return] :
    while ( Expression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return]
```

14.7.3.1 Static Semantics: Early Errors

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

- It is a Syntax Error if `IsLabelledFunction(Statement)` is **true**.

NOTE It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

14.7.3.2 Runtime Semantics: WhileLoopEvaluation

With parameter *labelSet*.

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

1. Let *V* be **undefined**.
2. Repeat,
 - a. Let *exprRef* be the result of evaluating *Expression*.
 - b. Let *exprValue* be ? `GetValue(exprRef)`.
 - c. If ! `ToBoolean(exprValue)` is **false**, return `NormalCompletion(V)`.
 - d. Let *stmtResult* be the result of evaluating *Statement*.
 - e. If `LoopContinues(stmtResult, labelSet)` is **false**, return `Completion(UpdateEmpty(stmtResult, V))`.
 - f. If *stmtResult*.[[Value]] is not empty, set *V* to *stmtResult*.[[Value]].

14.7.4 The for Statement

Syntax

*ForStatement*_[Yield, Await, Return] :

```
for ( [lookahead ≠ let [] ] Expression[~In, ?Yield, ?Await] opt ; Expression[+In, ?Yield, ?Await] opt ;  
      Expression[+In, ?Yield, ?Await] opt ) Statement[?Yield, ?Await, ?Return]  
for ( var VariableDeclarationList[~In, ?Yield, ?Await] ; Expression[+In, ?Yield, ?Await] opt ;  
      Expression[+In, ?Yield, ?Await] opt ) Statement[?Yield, ?Await, ?Return]  
for ( LexicalDeclaration[~In, ?Yield, ?Await] Expression[+In, ?Yield, ?Await] opt ;  
      Expression[+In, ?Yield, ?Await] opt ) Statement[?Yield, ?Await, ?Return]
```

14.7.4.1 Static Semantics: Early Errors

ForStatement :

```
for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement  
for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement  
for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement
```

- It is a Syntax Error if `IsLabelledFunction(Statement)` is **true**.

NOTE It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

ForStatement : **for** (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

- It is a Syntax Error if any element of the [BoundNames](#) of *LexicalDeclaration* also occurs in the [VarDeclaredNames](#) of *Statement*.

14.7.4.2 Runtime Semantics: ForLoopEvaluation

With parameter *labelSet*.

ForStatement : **for** (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. If the first *Expression* is present, then
 - a. Let *exprRef* be the result of evaluating the first *Expression*.
 - b. Perform ? [GetValue](#)(*exprRef*).
2. Return ? [ForBodyEvaluation](#)(the second *Expression*, the third *Expression*, *Statement*, « », *labelSet*).

ForStatement : **for** (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Let *varDcl* be the result of evaluating *VariableDeclarationList*.
2. [ReturnIfAbrupt](#)(*varDcl*).
3. Return ? [ForBodyEvaluation](#)(the first *Expression*, the second *Expression*, *Statement*, « », *labelSet*).

ForStatement : **for** (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

1. Let *oldEnv* be the [running execution context](#)'s [LexicalEnvironment](#).
2. Let *loopEnv* be [NewDeclarativeEnvironment](#)(*oldEnv*).
3. Let *isConst* be [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. Perform ! *loopEnv*.[CreateImmutableBinding](#)(*dn*, **true**).
 - b. Else,
 - i. Perform ! *loopEnv*.[CreateMutableBinding](#)(*dn*, **false**).
6. Set the [running execution context](#)'s [LexicalEnvironment](#) to *loopEnv*.
7. Let *forDcl* be the result of evaluating *LexicalDeclaration*.
8. If *forDcl* is an [abrupt completion](#), then
 - a. Set the [running execution context](#)'s [LexicalEnvironment](#) to *oldEnv*.
 - b. Return [Completion](#)(*forDcl*).
9. If *isConst* is **false**, let *perIterationLets* be *boundNames*; otherwise let *perIterationLets* be « ».
10. Let *bodyResult* be [ForBodyEvaluation](#)(the first *Expression*, the second *Expression*, *Statement*, *perIterationLets*, *labelSet*).
11. Set the [running execution context](#)'s [LexicalEnvironment](#) to *oldEnv*.
12. Return [Completion](#)(*bodyResult*).

14.7.4.3 ForBodyEvaluation (*test*, *increment*, *stmt*, *perIterationBindings*, *labelSet*)

The abstract operation [ForBodyEvaluation](#) takes arguments *test*, *increment*, *stmt*, *perIterationBindings*, and *labelSet*. It performs the following steps when called:

1. Let *V* be **undefined**.
2. Perform ? [CreatePerIterationEnvironment](#)(*perIterationBindings*).
3. Repeat,

- a. If *test* is not [empty], then
 - i. Let *testRef* be the result of evaluating *test*.
 - ii. Let *testValue* be ? *GetValue(testRef)*.
 - iii. If ! *ToBoolean(testValue)* is **false**, return *NormalCompletion(V)*.
- b. Let *result* be the result of evaluating *stmt*.
- c. If *LoopContinues(result, labelSet)* is **false**, return *Completion(UpdateEmpty(result, V))*.
- d. If *result.[[Value]]* is not empty, set *V* to *result.[[Value]]*.
- e. Perform ? *CreatePerIterationEnvironment(perIterationBindings)*.
- f. If *increment* is not [empty], then
 - i. Let *incRef* be the result of evaluating *increment*.
 - ii. Perform ? *GetValue(incRef)*.

14.7.4.4 CreatePerIterationEnvironment (*perIterationBindings*)

The abstract operation *CreatePerIterationEnvironment* takes argument *perIterationBindings*. It performs the following steps when called:

1. If *perIterationBindings* has any elements, then
 - a. Let *lastIterationEnv* be the **running execution context**'s *LexicalEnvironment*.
 - b. Let *outer* be *lastIterationEnv.[[OuterEnv]]*.
 - c. **Assert**: *outer* is not **null**.
 - d. Let *thisIterationEnv* be *NewDeclarativeEnvironment(outer)*.
 - e. For each element *bn* of *perIterationBindings*, do
 - i. Perform ! *thisIterationEnv.CreateMutableBinding(bn, false)*.
 - ii. Let *lastValue* be ? *lastIterationEnv.GetBindingValue(bn, true)*.
 - iii. Perform *thisIterationEnv.InitializeBinding(bn, lastValue)*.
 - f. Set the **running execution context**'s *LexicalEnvironment* to *thisIterationEnv*.
2. Return **undefined**.

14.7.5 The **for-in**, **for-of**, and **for-await-of** Statements

Syntax

```

ForInOfStatement [Yield, Await, Return] :
  for ( [lookahead ≠ let [ ] LeftHandSideExpression [?Yield, ?Await] in Expression [+In, ?Yield, ?Await] )
    Statement [?Yield, ?Await, ?Return]
  for ( var ForBinding [?Yield, ?Await] in Expression [+In, ?Yield, ?Await] )
    Statement [?Yield, ?Await, ?Return]
  for ( ForDeclaration [?Yield, ?Await] in Expression [+In, ?Yield, ?Await] )
    Statement [?Yield, ?Await, ?Return]
  for ( [lookahead ∉ { let, async of }] LeftHandSideExpression [?Yield, ?Await] of
    AssignmentExpression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return]
  for ( var ForBinding [?Yield, ?Await] of AssignmentExpression [+In, ?Yield, ?Await] )
    Statement [?Yield, ?Await, ?Return]
  for ( ForDeclaration [?Yield, ?Await] of AssignmentExpression [+In, ?Yield, ?Await] )
    Statement [?Yield, ?Await, ?Return]

```

```

[+Await] for await ( [lookahead ≠ let] LeftHandSideExpression [?Yield, ?Await] of
  AssignmentExpression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return]
[+Await] for await ( var ForBinding [?Yield, ?Await] of AssignmentExpression [+In, ?Yield, ?Await] )
  Statement [?Yield, ?Await, ?Return]
[+Await] for await ( ForDeclaration [?Yield, ?Await] of AssignmentExpression [+In, ?Yield, ?Await] )
  Statement [?Yield, ?Await, ?Return]

```

ForDeclaration [Yield, Await] :

LetOrConst ForBinding [?Yield, ?Await]

ForBinding [Yield, Await] :

BindingIdentifier [?Yield, ?Await]

BindingPattern [?Yield, ?Await]

NOTE This section is extended by Annex B.3.6.

14.7.5.1 Static Semantics: Early Errors

ForInOfStatement :

```

for ( LeftHandSideExpression in Expression ) Statement
for ( var ForBinding in Expression ) Statement
for ( ForDeclaration in Expression ) Statement
for ( LeftHandSideExpression of AssignmentExpression ) Statement
for ( var ForBinding of AssignmentExpression ) Statement
for ( ForDeclaration of AssignmentExpression ) Statement
for await ( LeftHandSideExpression of AssignmentExpression ) Statement
for await ( var ForBinding of AssignmentExpression ) Statement
for await ( ForDeclaration of AssignmentExpression ) Statement

```

- It is a Syntax Error if `IsLabelledFunction(Statement)` is **true**.

NOTE It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

ForInOfStatement :

```

for ( LeftHandSideExpression in Expression ) Statement
for ( LeftHandSideExpression of AssignmentExpression ) Statement
for await ( LeftHandSideExpression of AssignmentExpression ) Statement

```

If *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral*, the following Early Error rules are applied:

- It is a Syntax Error if *LeftHandSideExpression* is not **covering** an *AssignmentPattern*.
- All Early Error rules for *AssignmentPattern* and its derived productions also apply to the *AssignmentPattern* that is **covered** by *LeftHandSideExpression*.

If *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral*, the following Early Error rule is applied:

- It is a Syntax Error if `AssignmentTargetType` of *LeftHandSideExpression* is not **simple**.

ForInOfStatement :

for (*ForDeclaration* **in** *Expression*) *Statement*

for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

for await (*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*.
- It is a Syntax Error if the **BoundNames** of *ForDeclaration* contains any duplicate entries.

14.7.5.2 Static Semantics: **IsDestructuring**

MemberExpression : *PrimaryExpression*

1. If *PrimaryExpression* is either an *ObjectLiteral* or an *ArrayLiteral*, return **true**.
2. Return **false**.

MemberExpression :

MemberExpression [*Expression*]

MemberExpression . *IdentifierName*

MemberExpression *TemplateLiteral*

SuperProperty

MetaProperty

new *MemberExpression* *Arguments*

NewExpression :

new *NewExpression*

LeftHandSideExpression :

CallExpression

OptionalExpression

1. Return **false**.

ForDeclaration : *LetOrConst* *ForBinding*

1. Return **IsDestructuring** of *ForBinding*.

ForBinding : *BindingIdentifier*

1. Return **false**.

ForBinding : *BindingPattern*

1. Return **true**.

NOTE This section is extended by Annex B.3.6.

14.7.5.3 Runtime Semantics: **ForDeclarationBindingInitialization**

With parameters *value* and *environment*.

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 10.2.10). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

ForDeclaration : *LetOrConst ForBinding*

1. Return the result of performing **BindingInitialization** for *ForBinding* passing *value* and *environment* as the arguments.

14.7.5.4 Runtime Semantics: ForDeclarationBindingInstantiation

With parameter *environment*.

ForDeclaration : *LetOrConst ForBinding*

1. **Assert**: *environment* is a **declarative Environment Record**.
2. For each element *name* of the **BoundNames** of *ForBinding*, do
 - a. If **IsConstantDeclaration** of *LetOrConst* is **true**, then
 - i. Perform ! *environment*.**CreateImmutableBinding**(*name*, **true**).
 - b. Else,
 - i. Perform ! *environment*.**CreateMutableBinding**(*name*, **false**).

14.7.5.5 Runtime Semantics: ForInOfLoopEvaluation

With parameter *labelSet*.

ForInOfStatement : **for** (*LeftHandSideExpression in Expression*) *Statement*

1. Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« », *Expression*, **enumerate**).
2. Return ? **ForIn/OfBodyEvaluation**(*LeftHandSideExpression*, *Statement*, *keyResult*, **enumerate**, **assignment**, *labelSet*).

ForInOfStatement : **for** (**var** *ForBinding in Expression*) *Statement*

1. Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« », *Expression*, **enumerate**).
2. Return ? **ForIn/OfBodyEvaluation**(*ForBinding*, *Statement*, *keyResult*, **enumerate**, *varBinding*, *labelSet*).

ForInOfStatement : **for** (*ForDeclaration in Expression*) *Statement*

1. Let *keyResult* be ? **ForIn/OfHeadEvaluation**(**BoundNames** of *ForDeclaration*, *Expression*, **enumerate**).
2. Return ? **ForIn/OfBodyEvaluation**(*ForDeclaration*, *Statement*, *keyResult*, **enumerate**, **lexicalBinding**, *labelSet*).

ForInOfStatement : **for** (*LeftHandSideExpression of AssignmentExpression*) *Statement*

1. Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« », *AssignmentExpression*, **iterate**).
2. Return ? **ForIn/OfBodyEvaluation**(*LeftHandSideExpression*, *Statement*, *keyResult*, **iterate**, **assignment**, *labelSet*).

ForInOfStatement : **for** (**var** *ForBinding of AssignmentExpression*) *Statement*

1. Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« », *AssignmentExpression*, **iterate**).

- Return ? **ForIn/OfBodyEvaluation**(*ForBinding*, *Statement*, *keyResult*, *iterate*, *varBinding*, *labelSet*).

ForInOfStatement : **for** (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

- Let *keyResult* be ? **ForIn/OfHeadEvaluation**(**BoundNames** of *ForDeclaration*, *AssignmentExpression*, *iterate*).
- Return ? **ForIn/OfBodyEvaluation**(*ForDeclaration*, *Statement*, *keyResult*, *iterate*, *lexicalBinding*, *labelSet*).

ForInOfStatement : **for await** (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

- Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« » , *AssignmentExpression*, *async-iterate*).
- Return ? **ForIn/OfBodyEvaluation**(*LeftHandSideExpression*, *Statement*, *keyResult*, *iterate*, *assignment*, *labelSet*, *async*).

ForInOfStatement : **for await** (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

- Let *keyResult* be ? **ForIn/OfHeadEvaluation**(« » , *AssignmentExpression*, *async-iterate*).
- Return ? **ForIn/OfBodyEvaluation**(*ForBinding*, *Statement*, *keyResult*, *iterate*, *varBinding*, *labelSet*, *async*).

ForInOfStatement : **for await** (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

- Let *keyResult* be ? **ForIn/OfHeadEvaluation**(**BoundNames** of *ForDeclaration*, *AssignmentExpression*, *async-iterate*).
- Return ? **ForIn/OfBodyEvaluation**(*ForDeclaration*, *Statement*, *keyResult*, *iterate*, *lexicalBinding*, *labelSet*, *async*).

NOTE This section is extended by Annex B.3.6.

14.7.5.6 ForIn/OfHeadEvaluation (*uninitializedBoundNames*, *expr*, *iterationKind*)

The abstract operation ForIn/OfHeadEvaluation takes arguments *uninitializedBoundNames*, *expr*, and *iterationKind* (either *enumerate*, *iterate*, or *async-iterate*). It performs the following steps when called:

- Let *oldEnv* be the **running execution context**'s **LexicalEnvironment**.
- If *uninitializedBoundNames* is not an empty **List**, then
 - Assert**: *uninitializedBoundNames* has no duplicate entries.
 - Let *newEnv* be **NewDeclarativeEnvironment**(*oldEnv*).
 - For each String *name* of *uninitializedBoundNames*, do
 - Perform ! *newEnv*.**CreateMutableBinding**(*name*, **false**).
 - Set the **running execution context**'s **LexicalEnvironment** to *newEnv*.
- Let *exprRef* be the result of evaluating *expr*.
- Set the **running execution context**'s **LexicalEnvironment** to *oldEnv*.
- Let *exprValue* be ? **GetValue**(*exprRef*).
- If *iterationKind* is **enumerate**, then
 - If *exprValue* is **undefined** or **null**, then
 - Return **Completion** { **[[Type]]**: **break**, **[[Value]]**: **empty**, **[[Target]]**: **empty** }.
 - Let *obj* be ! **ToObject**(*exprValue*).
 - Let *iterator* be ? **EnumerateObjectProperties**(*obj*).
 - Let *nextMethod* be ! **GetV**(*iterator*, **"next"**).
 - Return the **Record** { **[[Iterator]]**: *iterator*, **[[NextMethod]]**: *nextMethod*, **[[Done]]**: **false** }.
- Else,

- a. **Assert:** *iterationKind* is `iterate` or `async-iterate`.
- b. If *iterationKind* is `async-iterate`, let *iteratorHint* be `async`.
- c. Else, let *iteratorHint* be `sync`.
- d. Return ? `GetIterator`(*exprValue*, *iteratorHint*).

14.7.5.7 ForIn/OfBodyEvaluation (*lhs*, *stmt*, *iteratorRecord*, *iterationKind*, *lhsKind*, *labelSet* [, *iteratorKind*])

The abstract operation ForIn/OfBodyEvaluation takes arguments *lhs*, *stmt*, *iteratorRecord*, *iterationKind*, *lhsKind* (either `assignment`, `varBinding` or `lexicalBinding`), and *labelSet* and optional argument *iteratorKind* (either `sync` or `async`). It performs the following steps when called:

1. If *iterationKind* is not present, set *iterationKind* to `sync`.
2. Let *oldEnv* be the `running execution context`'s `LexicalEnvironment`.
3. Let *V* be `undefined`.
4. Let *destructuring* be `IsDestructuring` of *lhs*.
5. If *destructuring* is `true` and if *lhsKind* is `assignment`, then
 - a. **Assert:** *lhs* is a `LeftHandSideExpression`.
 - b. Let *assignmentPattern* be the `AssignmentPattern` that is `covered` by *lhs*.
6. Repeat,
 - a. Let *nextResult* be ? `Call`(*iteratorRecord*.[[`NextMethod`]], *iteratorRecord*.[[`Iterator`]]).
 - b. If *iterationKind* is `async`, set *nextResult* to ? `Await`(*nextResult*).
 - c. If `Type`(*nextResult*) is not `Object`, throw a `TypeError` exception.
 - d. Let *done* be ? `IteratorComplete`(*nextResult*).
 - e. If *done* is `true`, return `NormalCompletion`(*V*).
 - f. Let *nextValue* be ? `IteratorValue`(*nextResult*).
 - g. If *lhsKind* is either `assignment` or `varBinding`, then
 - i. If *destructuring* is `false`, then
 1. Let *lhsRef* be the result of evaluating *lhs*. (It may be evaluated repeatedly.)
 - h. Else,
 - i. **Assert:** *lhsKind* is `lexicalBinding`.
 - ii. **Assert:** *lhs* is a `ForDeclaration`.
 - iii. Let *iterationEnv* be `NewDeclarativeEnvironment`(*oldEnv*).
 - iv. Perform `ForDeclarationBindingInstantiation` for *lhs* passing *iterationEnv* as the argument.
 - v. Set the `running execution context`'s `LexicalEnvironment` to *iterationEnv*.
 - vi. If *destructuring* is `false`, then
 1. **Assert:** *lhs* binds a single name.
 2. Let *lhsName* be the sole element of `BoundNames` of *lhs*.
 3. Let *lhsRef* be ! `ResolveBinding`(*lhsName*).
 - i. If *destructuring* is `false`, then
 - i. If *lhsRef* is an `abrupt completion`, then
 1. Let *status* be *lhsRef*.
 - ii. Else if *lhsKind* is `lexicalBinding`, then
 1. Let *status* be `InitializeReferencedBinding`(*lhsRef*, *nextValue*).
 - iii. Else,
 1. Let *status* be `PutValue`(*lhsRef*, *nextValue*).
 - j. Else,
 - i. If *lhsKind* is `assignment`, then

1. Let *status* be **DestructuringAssignmentEvaluation** of *assignmentPattern* with argument *nextValue*.
- ii. Else if *lhsKind* is **varBinding**, then
 1. **Assert**: *lhs* is a *ForBinding*.
 2. Let *status* be **BindingInitialization** of *lhs* with arguments *nextValue* and **undefined**.
- iii. Else,
 1. **Assert**: *lhsKind* is **lexicalBinding**.
 2. **Assert**: *lhs* is a *ForDeclaration*.
 3. Let *status* be **ForDeclarationBindingInitialization** of *lhs* with arguments *nextValue* and *iterationEnv*.
- k. If *status* is an **abrupt completion**, then
 - i. Set the **running execution context**'s **LexicalEnvironment** to *oldEnv*.
 - ii. If *iteratorKind* is **async**, return ? **AsyncIteratorClose**(*iteratorRecord*, *status*).
 - iii. If *iterationKind* is **enumerate**, then
 1. Return *status*.
 - iv. Else,
 1. **Assert**: *iterationKind* is **iterate**.
 2. Return ? **IteratorClose**(*iteratorRecord*, *status*).
- l. Let *result* be the result of evaluating *stmt*.
- m. Set the **running execution context**'s **LexicalEnvironment** to *oldEnv*.
- n. If **LoopContinues**(*result*, *labelSet*) is **false**, then
 - i. If *iterationKind* is **enumerate**, then
 1. Return **Completion**(**UpdateEmpty**(*result*, *V*)).
 - ii. Else,
 1. **Assert**: *iterationKind* is **iterate**.
 2. Set *status* to **UpdateEmpty**(*result*, *V*).
 3. If *iteratorKind* is **async**, return ? **AsyncIteratorClose**(*iteratorRecord*, *status*).
 4. Return ? **IteratorClose**(*iteratorRecord*, *status*).
- o. If *result*.[[Value]] is not empty, set *V* to *result*.[[Value]].

14.7.5.8 Runtime Semantics: Evaluation

ForBinding : *BindingIdentifier*

1. Let *bindingId* be **StringValue** of *BindingIdentifier*.
2. Return ? **ResolveBinding**(*bindingId*).

14.7.5.9 EnumerateObjectProperties (*O*)

The abstract operation **EnumerateObjectProperties** takes argument *O*. It performs the following steps when called:

1. **Assert**: **Type**(*O*) is **Object**.
2. Return an **Iterator** object (27.1.1.2) whose **next** method iterates over all the **String**-valued keys of enumerable properties of *O*. The iterator object is never directly accessible to ECMAScript code. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.

The iterator's **throw** and **return** methods are **null** and are never invoked. The iterator's **next** method processes object properties to determine whether the property key should be returned as an iterator value. Returned property keys do not include keys that are **Symbols**. Properties of the target object 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 target object 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 the target object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not 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 enumerable property names of prototype objects must be obtained by invoking `EnumerateObjectProperties` passing the prototype object as the argument. `EnumerateObjectProperties` must obtain the own property keys of the target object by calling its `[[OwnPropertyKeys]]` internal method. Property attributes of the target object must be obtained by calling its `[[GetOwnProperty]]` internal method.

In addition, if neither `O` nor any object in its prototype chain is a [Proxy exotic object](#), [Integer-Indexed exotic object](#), [module namespace exotic object](#), or implementation provided [exotic object](#), then the iterator must behave as would the iterator given by `CreateForInIterator(O)` until one of the following occurs:

- the value of the `[[Prototype]]` internal slot of `O` or an object in its prototype chain changes,
- a property is removed from `O` or an object in its prototype chain,
- a property is added to an object in `O`'s prototype chain, or
- the value of the `[[Enumerable]]` attribute of a property of `O` or an object in its prototype chain changes.

NOTE 1

ECMAScript implementations are not required to implement the algorithm in [14.7.5.10.2.1](#) directly. They may choose any implementation whose behaviour will not deviate from that algorithm unless one of the constraints in the previous paragraph is violated.

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

```
function* EnumerateObjectProperties(obj) {
  const visited = new Set();
  for (const key of Reflect.ownKeys(obj)) {
    if (typeof key === "symbol") continue;
    const desc = Reflect.getOwnPropertyDescriptor(obj, key);
    if (desc) {
      visited.add(key);
      if (desc.enumerable) yield key;
    }
  }
  const proto = Reflect.getPrototypeOf(obj);
  if (proto === null) return;
  for (const protoKey of EnumerateObjectProperties(proto)) {
    if (!visited.has(protoKey)) yield protoKey;
  }
}
```

NOTE 2

The list of exotic objects for which implementations are not required to match `CreateForInIterator` was chosen because implementations historically differed in behaviour for those cases, and agreed in all others.

14.7.5.10 For-In Iterator Objects

A For-In Iterator is an object that represents a specific iteration over some specific object. For-In Iterator objects are never directly accessible to ECMAScript code; they exist solely to illustrate the behaviour of `EnumerateObjectProperties`.

14.7.5.10.1 `CreateForInIterator (object)`

The abstract operation `CreateForInIterator` takes argument *object*. It is used to create a For-In Iterator object which iterates over the own and inherited enumerable string properties of *object* in a specific order. It performs the following steps when called:

1. **Assert:** `Type(object)` is Object.
2. Let *iterator* be ! `OrdinaryObjectCreate(%ForInIteratorPrototype%, « [[Object]], [[ObjectWasVisited]], [[VisitedKeys]], [[RemainingKeys]] »)`.
3. Set *iterator*.[[Object]] to *object*.
4. Set *iterator*.[[ObjectWasVisited]] to **false**.
5. Set *iterator*.[[VisitedKeys]] to a new empty List.
6. Set *iterator*.[[RemainingKeys]] to a new empty List.
7. Return *iterator*.

14.7.5.10.2 The `%ForInIteratorPrototype%` Object

The `%ForInIteratorPrototype%` object:

- has properties that are inherited by all For-In Iterator Objects.
- is an **ordinary object**.
- has a [[Prototype]] internal slot whose value is `%IteratorPrototype%`.
- is never directly accessible to ECMAScript code.
- has the following properties:

14.7.5.10.2.1 `%ForInIteratorPrototype%.next ()`

1. Let *O* be the **this** value.
2. **Assert:** `Type(O)` is Object.
3. **Assert:** *O* has all of the internal slots of a For-In Iterator Instance (14.7.5.10.3).
4. Let *object* be *O*.[[Object]].
5. Let *visited* be *O*.[[VisitedKeys]].
6. Let *remaining* be *O*.[[RemainingKeys]].
7. Repeat,
 - a. If *O*.[[ObjectWasVisited]] is **false**, then
 - i. Let *keys* be ? *object*.[[OwnPropertyKeys]]().
 - ii. For each element *key* of *keys*, do
 1. If `Type(key)` is String, then
 - a. Append *key* to *remaining*.

- iii. Set $O.[[ObjectWasVisited]]$ to **true**.
- b. Repeat, while *remaining* is not empty,
 - i. Let r be the first element of *remaining*.
 - ii. Remove the first element from *remaining*.
 - iii. If there does not exist an element v of *visited* such that $\text{SameValue}(r, v)$ is **true**, then
 - 1. Let $desc$ be $? object.[[GetOwnProperty]](r)$.
 - 2. If $desc$ is not **undefined**, then
 - a. Append r to *visited*.
 - b. If $desc.[[Enumerable]]$ is **true**, return $\text{CreateIterResultObject}(r, \text{false})$.
- c. Set $object$ to $? object.[[GetPrototypeOf]]()$.
- d. Set $O.[[Object]]$ to $object$.
- e. Set $O.[[ObjectWasVisited]]$ to **false**.
- f. If $object$ is **null**, return $\text{CreateIterResultObject}(\text{undefined}, \text{true})$.

14.7.5.10.3 Properties of For-In Iterator Instances

For-In Iterator instances are ordinary objects that inherit properties from the `%ForInIteratorPrototype%` intrinsic object. For-In Iterator instances are initially created with the internal slots listed in Table 38.

Table 38: Internal Slots of For-In Iterator Instances

Internal Slot	Description
<code>[[Object]]</code>	The Object value whose properties are being iterated.
<code>[[ObjectWasVisited]]</code>	true if the iterator has invoked <code>[[OwnPropertyKeys]]</code> on <code>[[Object]]</code> , false otherwise.
<code>[[VisitedKeys]]</code>	A list of String values which have been emitted by this iterator thus far.
<code>[[RemainingKeys]]</code>	A list of String values remaining to be emitted for the current object, before iterating the properties of its prototype (if its prototype is not null).

14.8 The `continue` Statement

Syntax

```
ContinueStatement[Yield, Await] :
    continue ;
    continue [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;
```

14.8.1 Static Semantics: Early Errors

```
ContinueStatement :
    continue ;
    continue LabelIdentifier ;
```

- It is a Syntax Error if this *ContinueStatement* is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.

14.8.2 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* }.

14.9 The break Statement

Syntax

*BreakStatement*_[Yield, Await] :

break ;

break [no *LineTerminator* here] *LabelIdentifier*_[?Yield, ?Await] ;

14.9.1 Static Semantics: Early Errors

BreakStatement : **break** ;

- It is a Syntax Error if this *BreakStatement* is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement* or a *SwitchStatement*.

14.9.2 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* }.

14.10 The return Statement

Syntax

*ReturnStatement*_[Yield, Await] :

return ;

return [no *LineTerminator* here] *Expression*_[+In, ?Yield, ?Await] ;

NOTE

A **return** statement causes a function to cease execution and, in most cases, returns a value to the caller. If *Expression* is omitted, the return value is **undefined**. Otherwise, the return value is the value of *Expression*. A **return** statement may not actually return a value to the caller depending on surrounding context. For example, in a **try** block, a **return** statement's completion record may be replaced with another completion record during evaluation of the **finally** block.

14.10.1 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. If ! **GetGeneratorKind**() is **async**, set *exprValue* to ? **Await**(*exprValue*).
4. Return **Completion** { [[Type]]: return, [[Value]]: *exprValue*, [[Target]]: empty }.

14.11 The **with** Statement

Syntax

*WithStatement*_[Yield, Await, Return] :
with (*Expression*_[+In, ?Yield, ?Await]) *Statement*_[?Yield, ?Await, ?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.

14.11.1 Static Semantics: Early Errors

WithStatement : **with** (*Expression*) *Statement*

- It is a Syntax Error if the code that matches this production is contained in **strict mode 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.

14.11.2 Runtime Semantics: Evaluation

WithStatement : **with** (*Expression*) *Statement*

1. Let *val* be the result of evaluating *Expression*.
2. Let *obj* be ? **ToObject**(? **GetValue**(*val*)).
3. Let *oldEnv* be the **running execution context**'s **LexicalEnvironment**.

4. Let *newEnv* be `NewObjectEnvironment(obj, oldEnv)`.
5. Set the *withEnvironment* flag of *newEnv* to **true**.
6. Set the **running execution context**'s `LexicalEnvironment` to *newEnv*.
7. Let *C* be the result of evaluating *Statement*.
8. Set the **running execution context**'s `LexicalEnvironment` to *oldEnv*.
9. Return `Completion(UpdateEmpty(C, undefined))`.

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.

14.12 The switch Statement

Syntax

```

SwitchStatement[Yield, Await, Return] :
    switch ( Expression [+In, ?Yield, ?Await] ) CaseBlock[?Yield, ?Await, ?Return]

CaseBlock[Yield, Await, Return] :
    { CaseClauses[?Yield, ?Await, ?Return] opt }
    { CaseClauses[?Yield, ?Await, ?Return] opt DefaultClause[?Yield, ?Await, ?Return]
      CaseClauses[?Yield, ?Await, ?Return] opt }

CaseClauses[Yield, Await, Return] :
    CaseClause[?Yield, ?Await, ?Return]
    CaseClauses[?Yield, ?Await, ?Return] CaseClause[?Yield, ?Await, ?Return]

CaseClause[Yield, Await, Return] :
    case Expression [+In, ?Yield, ?Await] : StatementList[?Yield, ?Await, ?Return] opt

DefaultClause[Yield, Await, Return] :
    default : StatementList[?Yield, ?Await, ?Return] opt

```

14.12.1 Static Semantics: Early Errors

SwitchStatement : `switch (Expression) CaseBlock`

- It is a Syntax Error if the `LexicallyDeclaredNames` of *CaseBlock* contains any duplicate entries.
- It is a Syntax Error if any element of the `LexicallyDeclaredNames` of *CaseBlock* also occurs in the `VarDeclaredNames` of *CaseBlock*.

14.12.2 Runtime Semantics: CaseBlockEvaluation

With parameter *input*.

CaseBlock : { }

1. Return `NormalCompletion(undefined)`.

CaseBlock : { *CaseClauses* }

1. Let *V* be **undefined**.
2. Let *A* be the **List** of *CaseClause* items in *CaseClauses*, in source text order.
3. Let *found* be **false**.
4. For each *CaseClause* *C* of *A*, do
 - a. If *found* is **false**, then
 - i. Set *found* to ? **CaseClauseIsSelected**(*C*, *input*).
 - b. If *found* is **true**, then
 - i. Let *R* be the result of evaluating *C*.
 - ii. If *R*.[[Value]] is not **empty**, set *V* to *R*.[[Value]].
 - iii. If *R* is an **abrupt completion**, return **Completion**(**UpdateEmpty**(*R*, *V*)).
5. Return **NormalCompletion**(*V*).

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. Let *V* be **undefined**.
2. If the first *CaseClauses* is present, then
 - a. Let *A* be the **List** of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Else,
 - a. Let *A* be « ».
4. Let *found* be **false**.
5. For each *CaseClause* *C* of *A*, do
 - a. If *found* is **false**, then
 - i. Set *found* to ? **CaseClauseIsSelected**(*C*, *input*).
 - b. If *found* is **true**, then
 - i. Let *R* be the result of evaluating *C*.
 - ii. If *R*.[[Value]] is not **empty**, set *V* to *R*.[[Value]].
 - iii. If *R* is an **abrupt completion**, return **Completion**(**UpdateEmpty**(*R*, *V*)).
6. Let *foundInB* be **false**.
7. If the second *CaseClauses* is present, then
 - a. Let *B* be the **List** of *CaseClause* items in the second *CaseClauses*, in source text order.
8. Else,
 - a. Let *B* be « ».
9. If *found* is **false**, then
 - a. For each *CaseClause* *C* of *B*, do
 - i. If *foundInB* is **false**, then
 1. Set *foundInB* to ? **CaseClauseIsSelected**(*C*, *input*).
 - ii. If *foundInB* is **true**, then
 1. Let *R* be the result of evaluating *CaseClause* *C*.
 2. If *R*.[[Value]] is not **empty**, set *V* to *R*.[[Value]].
 3. If *R* is an **abrupt completion**, return **Completion**(**UpdateEmpty**(*R*, *V*)).
10. If *foundInB* is **true**, return **NormalCompletion**(*V*).
11. Let *R* be the result of evaluating *DefaultClause*.
12. If *R*.[[Value]] is not **empty**, set *V* to *R*.[[Value]].
13. If *R* is an **abrupt completion**, return **Completion**(**UpdateEmpty**(*R*, *V*)).
14. NOTE: The following is another complete iteration of the second *CaseClauses*.
15. For each *CaseClause* *C* of *B*, do

- a. Let R be the result of evaluating *CaseClause* C .
 - b. If $R.[[Value]]$ is not empty, set V to $R.[[Value]]$.
 - c. If R is an abrupt completion, return $Completion(UpdateEmpty(R, V))$.
16. Return $NormalCompletion(V)$.

14.12.3 CaseClauseIsSelected ($C, input$)

The abstract operation *CaseClauseIsSelected* takes arguments C (a *Parse Node* for *CaseClause*) and *input* (an *ECMAScript language value*). It determines whether C matches *input*. It performs the following steps when called:

1. **Assert:** C is an instance of the production *CaseClause* : **case** *Expression* : *StatementList*_{opt} .
2. Let *exprRef* be the result of evaluating the *Expression* of C .
3. Let *clauseSelector* be ? *GetValue*(*exprRef*).
4. Return the result of performing *Strict Equality Comparison* $input === clauseSelector$.

NOTE This operation does not execute C 's *StatementList* (if any). The *CaseBlock* algorithm uses its return value to determine which *StatementList* to start executing.

14.12.4 Runtime Semantics: Evaluation

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *switchValue* be ? *GetValue*(*exprRef*).
3. Let *oldEnv* be the *running execution context*'s *LexicalEnvironment*.
4. Let *blockEnv* be *NewDeclarativeEnvironment*(*oldEnv*).
5. Perform *BlockDeclarationInstantiation*(*CaseBlock*, *blockEnv*).
6. Set the *running execution context*'s *LexicalEnvironment* to *blockEnv*.
7. Let R be *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*.

14.13 Labelled Statements

Syntax

*LabelledStatement*_[Yield, Await, Return] :
*LabelIdentifier*_[?Yield, ?Await] : *LabelledItem*_[?Yield, ?Await, ?Return]

*LabelledItem*_[Yield, Await, Return] :
*Statement*_[?Yield, ?Await, ?Return]
*FunctionDeclaration*_[?Yield, ?Await, ~Default]

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.

14.13.1 Static Semantics: Early Errors

LabelledItem : *FunctionDeclaration*

- It is a Syntax Error if any source text matches this rule.

NOTE

An alternative definition for this rule is provided in B.3.2.

14.13.2 Static Semantics: IsLabelledFunction (*stmt*)

The abstract operation IsLabelledFunction takes argument *stmt*. It performs the following steps when called:

1. If *stmt* is not a *LabelledStatement*, return **false**.
2. Let *item* be the *LabelledItem* of *stmt*.
3. If *item* is *LabelledItem* : *FunctionDeclaration* , return **true**.
4. Let *subStmt* be the *Statement* of *item*.
5. Return *IsLabelledFunction*(*subStmt*).

14.13.3 Runtime Semantics: Evaluation

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *newLabelSet* be a new empty *List*.
2. Return *LabelledEvaluation* of this *LabelledStatement* with argument *newLabelSet*.

14.13.4 Runtime Semantics: LabelledEvaluation

With parameter *labelSet*.

BreakableStatement : *IterationStatement*

1. Let *stmtResult* be **LoopEvaluation** of *IterationStatement* with argument *labelSet*.
2. If *stmtResult*.[[Type]] is **break**, then
 - a. If *stmtResult*.[[Target]] is **empty**, then
 - i. If *stmtResult*.[[Value]] is **empty**, set *stmtResult* to **NormalCompletion(undefined)**.
 - ii. Else, set *stmtResult* to **NormalCompletion(stmtResult.[[Value]])**.
3. Return **Completion(stmtResult)**.

BreakableStatement : *SwitchStatement*

1. Let *stmtResult* be the result of evaluating *SwitchStatement*.
2. If *stmtResult*.[[Type]] is **break**, then
 - a. If *stmtResult*.[[Target]] is **empty**, then
 - i. If *stmtResult*.[[Value]] is **empty**, set *stmtResult* to **NormalCompletion(undefined)**.
 - ii. Else, set *stmtResult* to **NormalCompletion(stmtResult.[[Value]])**.
3. Return **Completion(stmtResult)**.

NOTE 1 A *BreakableStatement* is one that can be exited via an unlabelled *BreakStatement*.

LabelledStatement : *LabelIdentifier* : *LabelledItem*

1. Let *label* be the **StringValue** of *LabelIdentifier*.
2. Append *label* as an element of *labelSet*.
3. Let *stmtResult* be **LabelledEvaluation** of *LabelledItem* with argument *labelSet*.
4. If *stmtResult*.[[Type]] is **break** and **SameValue(stmtResult.[[Target]], *label*)** is **true**, then
 - a. Set *stmtResult* to **NormalCompletion(stmtResult.[[Value]])**.
5. Return **Completion(stmtResult)**.

LabelledItem : *FunctionDeclaration*

1. Return the result of evaluating *FunctionDeclaration*.

Statement :

BlockStatement
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
ThrowStatement
TryStatement
DebuggerStatement

1. Return the result of evaluating *Statement*.

The only two productions of *Statement* which have special semantics for *LabelledEvaluation* are *BreakableStatement* and *LabelledStatement*.

14.14 The **throw** Statement

Syntax

```
ThrowStatement[Yield, Await] :
    throw [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;
```

14.14.1 Runtime Semantics: Evaluation

ThrowStatement : **throw** *Expression* ;

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ? *GetValue*(*exprRef*).
3. Return *ThrowCompletion*(*exprValue*).

14.15 The **try** Statement

Syntax

```
TryStatement[Yield, Await, Return] :
    try Block[?Yield, ?Await, ?Return] Catch[?Yield, ?Await, ?Return]
    try Block[?Yield, ?Await, ?Return] Finally[?Yield, ?Await, ?Return]
    try Block[?Yield, ?Await, ?Return] Catch[?Yield, ?Await, ?Return] Finally[?Yield, ?Await, ?Return]
```

```
Catch[Yield, Await, Return] :
    catch ( CatchParameter[?Yield, ?Await] ) Block[?Yield, ?Await, ?Return]
    catch Block[?Yield, ?Await, ?Return]
```

```
Finally[Yield, Await, Return] :
    finally Block[?Yield, ?Await, ?Return]
```

```
CatchParameter[Yield, Await] :
    BindingIdentifier[?Yield, ?Await]
    BindingPattern[?Yield, ?Await]
```

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.

14.15.1 Static Semantics: Early Errors

Catch : **catch** (*CatchParameter*) *Block*

- It is a Syntax Error if **BoundNames** of *CatchParameter* contains any duplicate elements.
- 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.

14.15.2 Runtime Semantics: CatchClauseEvaluation

With parameter *thrownValue*.

Catch : **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. Perform ! **catchEnv**.**CreateMutableBinding**(*argName*, **false**).
4. Set the **running execution context**'s **LexicalEnvironment** to *catchEnv*.
5. Let *status* be **BindingInitialization** of *CatchParameter* with arguments *thrownValue* and *catchEnv*.
6. If *status* is an **abrupt completion**, then
 - a. Set the **running execution context**'s **LexicalEnvironment** to *oldEnv*.
 - b. Return **Completion**(*status*).
7. Let *B* be the result of evaluating *Block*.
8. Set the **running execution context**'s **LexicalEnvironment** to *oldEnv*.
9. Return **Completion**(*B*).

Catch : **catch** *Block*

1. Return the result of evaluating *Block*.

NOTE No matter how control leaves the *Block* the **LexicalEnvironment** is always restored to its former state.

14.15.3 Runtime Semantics: Evaluation

TryStatement : **try** *Block* *Catch*

1. Let *B* be the result of evaluating *Block*.
2. If *B*.**[[Type]]** is **throw**, let *C* be **CatchClauseEvaluation** of *Catch* with argument *B*.**[[Value]]**.
3. Else, let *C* be *B*.
4. Return **Completion**(**UpdateEmpty**(*C*, **undefined**)).

TryStatement : **try** *Block* *Finally*

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.

3. If $F.[[Type]]$ is normal, set F to B .
4. Return `Completion(UpdateEmpty(F , undefined))`.

TryStatement : **try** *Block* *Catch* *Finally*

1. Let B be the result of evaluating *Block*.
2. If $B.[[Type]]$ is throw, let C be `CatchClauseEvaluation` of *Catch* with argument $B.[[Value]]$.
3. Else, let C be B .
4. Let F be the result of evaluating *Finally*.
5. If $F.[[Type]]$ is normal, set F to C .
6. Return `Completion(UpdateEmpty(F , undefined))`.

14.16 The debugger Statement

Syntax

DebuggerStatement :
debugger ;

14.16.1 Runtime Semantics: Evaluation

NOTE Evaluating a *DebuggerStatement* 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*.

15 ECMAScript Language: Functions and Classes

NOTE Various ECMAScript language elements cause the creation of ECMAScript function objects (10.2). Evaluation of such functions starts with the execution of their `[[Call]]` internal method (10.2.1).

15.1 Parameter Lists

Syntax

*UniqueFormalParameters*_[*Yield*, *Await*] :

*FormalParameters*_[?Yield, ?Await]

*FormalParameters*_[Yield, Await] :

[empty]

*FunctionRestParameter*_[?Yield, ?Await]

*FormalParameterList*_[?Yield, ?Await]

*FormalParameterList*_[?Yield, ?Await] ,

*FormalParameterList*_[?Yield, ?Await] , *FunctionRestParameter*_[?Yield, ?Await]

*FormalParameterList*_[Yield, Await] :

*FormalParameter*_[?Yield, ?Await]

*FormalParameterList*_[?Yield, ?Await] , *FormalParameter*_[?Yield, ?Await]

*FunctionRestParameter*_[Yield, Await] :

*BindingRestElement*_[?Yield, ?Await]

*FormalParameter*_[Yield, Await] :

*BindingElement*_[?Yield, ?Await]

15.1.1 Static Semantics: Early Errors

UniqueFormalParameters : *FormalParameters*

- It is a Syntax Error if **BoundNames** of *FormalParameters* contains any duplicate elements.

FormalParameters : *FormalParameterList*

- It is a Syntax Error if **IsSimpleParameterList** of *FormalParameterList* is **false** and **BoundNames** of *FormalParameterList* contains any duplicate elements.

NOTE

Multiple occurrences of the same *BindingIdentifier* in a *FormalParameterList* is only allowed for functions which have simple parameter lists and which are not defined in **strict mode code**.

15.1.2 Static Semantics: ContainsExpression

ObjectBindingPattern :

{ }

{ *BindingRestProperty* }

1. Return **false**.

ObjectBindingPattern : { *BindingPropertyList* , *BindingRestProperty* }

1. Return **ContainsExpression** of *BindingPropertyList*.

ArrayBindingPattern : [*Elision*_{opt}]

1. Return **false**.

ArrayBindingPattern : [*Elision_{opt}* *BindingRestElement*]

1. Return **ContainsExpression** of *BindingRestElement*.

ArrayBindingPattern : [*BindingElementList* , *Elision_{opt}*]

1. Return **ContainsExpression** of *BindingElementList*.

ArrayBindingPattern : [*BindingElementList* , *Elision_{opt}* *BindingRestElement*]

1. Let *has* be **ContainsExpression** of *BindingElementList*.
2. If *has* is **true**, return **true**.
3. Return **ContainsExpression** of *BindingRestElement*.

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 : *Elision_{opt}* *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 **ContainsExpression** of *BindingElement*.

BindingElement : *BindingPattern* *Initializer*

1. Return **true**.

SingleNameBinding : *BindingIdentifier*

1. Return **false**.

SingleNameBinding : *BindingIdentifier* *Initializer*

1. Return **true**.

BindingRestElement : ... *BindingIdentifier*

1. Return **false**.

BindingRestElement : ... *BindingPattern*

1. Return **ContainsExpression** of *BindingPattern*.

FormalParameters : [empty]

1. Return **false**.

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. If **ContainsExpression** of *FormalParameterList* is **true**, return **true**.
2. Return **ContainsExpression** of *FunctionRestParameter*.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. If **ContainsExpression** of *FormalParameterList* is **true**, return **true**.
2. Return **ContainsExpression** of *FormalParameter*.

ArrowParameters : *BindingIdentifier*

1. Return **false**.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be **CoveredFormalsList** of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return **ContainsExpression** of *formals*.

AsyncArrowBindingIdentifier : *BindingIdentifier*

1. Return **false**.

15.1.3 Static Semantics: **IsSimpleParameterList**

BindingElement : *BindingPattern*

1. Return **false**.

BindingElement : *BindingPattern* *Initializer*

1. Return **false**.

SingleNameBinding : *BindingIdentifier*

1. Return **true**.

SingleNameBinding : *BindingIdentifier* *Initializer*

1. Return **false**.

FormalParameters : [empty]

1. Return **true**.

FormalParameters : *FunctionRestParameter*

1. Return **false**.

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. Return **false**.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. If *IsSimpleParameterList* of *FormalParameterList* is **false**, return **false**.
2. Return *IsSimpleParameterList* of *FormalParameter*.

FormalParameter : *BindingElement*

1. Return *IsSimpleParameterList* of *BindingElement*.

ArrowParameters : *BindingIdentifier*

1. Return **true**.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be *CoveredFormalsList* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return *IsSimpleParameterList* of *formals*.

AsyncArrowBindingIdentifier [*Yield*] : *BindingIdentifier* [*?Yield*, *+Await*]

1. Return **true**.

CoverCallExpressionAndAsyncArrowHead : *MemberExpression Arguments*

1. Let *head* be *CoveredAsyncArrowHead* of *CoverCallExpressionAndAsyncArrowHead*.
2. Return *IsSimpleParameterList* of *head*.

15.1.4 Static Semantics: HasInitializer

BindingElement : *BindingPattern*

1. Return **false**.

BindingElement : *BindingPattern Initializer*

1. Return **true**.

SingleNameBinding : *BindingIdentifier*

1. Return **false**.

SingleNameBinding : *BindingIdentifier Initializer*

1. Return **true**.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. If *HasInitializer* of *FormalParameterList* is **true**, return **true**.
2. Return *HasInitializer* of *FormalParameter*.

15.1.5 Static Semantics: ExpectedArgumentCount

FormalParameters :

[empty]

FunctionRestParameter

1. Return 0.

FormalParameters : *FormalParameterList* , *FunctionRestParameter*

1. Return [ExpectedArgumentCount](#) of *FormalParameterList*.

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.

FormalParameterList : *FormalParameter*

1. If [HasInitializer](#) of *FormalParameter* is **true**, return 0.
2. Return 1.

FormalParameterList : *FormalParameterList* , *FormalParameter*

1. Let *count* be [ExpectedArgumentCount](#) of *FormalParameterList*.
2. If [HasInitializer](#) of *FormalParameterList* is **true** or [HasInitializer](#) of *FormalParameter* is **true**, return *count*.
3. Return *count* + 1.

ArrowParameters : *BindingIdentifier*

1. Return 1.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be [CoveredFormalsList](#) of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return [ExpectedArgumentCount](#) of *formals*.

PropertySetParameterList : *FormalParameter*

1. If [HasInitializer](#) of *FormalParameter* is **true**, return 0.
2. Return 1.

AsyncArrowBindingIdentifier : *BindingIdentifier*

1. Return 1.

15.2 Function Definitions

Syntax

```

FunctionDeclaration [Yield, Await, Default] :
    function BindingIdentifier [?Yield, ?Await] ( FormalParameters [~Yield, ~Await] ) {
        FunctionBody [~Yield, ~Await] }
    [+Default] function ( FormalParameters [~Yield, ~Await] ) { FunctionBody [~Yield, ~Await] }

```

FunctionExpression :

```
function BindingIdentifier [~Yield, ~Await] opt ( FormalParameters [~Yield, ~Await] ) {
  FunctionBody [~Yield, ~Await] }
```

```
FunctionBody [Yield, Await] :
  FunctionStatementList [?Yield, ?Await]
```

```
FunctionStatementList [Yield, Await] :
  StatementList [?Yield, ?Await, +Return] opt
```

15.2.1 Static Semantics: Early Errors

FunctionDeclaration :

```
function BindingIdentifier ( FormalParameters ) { FunctionBody }
function ( FormalParameters ) { FunctionBody }
```

FunctionExpression :

```
function BindingIdentifieropt ( FormalParameters ) { FunctionBody }
```

- If the source code matching *FormalParameters* is **strict mode code**, the Early Error rules for *UniqueFormalParameters* : *FormalParameters* are applied.
- If *BindingIdentifier* is present and the source code matching *BindingIdentifier* is **strict mode code**, it is a Syntax Error if the **StringValue** of *BindingIdentifier* is "eval" or "arguments".
- It is a Syntax Error if **FunctionBodyContainsUseStrict** of *FunctionBody* is **true** and **IsSimpleParameterList** of *FormalParameters* is **false**.
- It is a Syntax Error if any element of the **BoundNames** of *FormalParameters* also occurs in the **LexicallyDeclaredNames** of *FunctionBody*.
- It is a Syntax Error if *FormalParameters* **Contains SuperProperty** is **true**.
- It is a Syntax Error if *FunctionBody* **Contains SuperProperty** is **true**.
- It is a Syntax Error if *FormalParameters* **Contains SuperCall** is **true**.
- It is a Syntax Error if *FunctionBody* **Contains SuperCall** is **true**.

NOTE The **LexicallyDeclaredNames** of a *FunctionBody* does not include identifiers bound using var or function declarations.

FunctionBody : *FunctionStatementList*

- It is a Syntax Error if the **LexicallyDeclaredNames** of *FunctionStatementList* contains any duplicate entries.
- It is a Syntax Error if any element of the **LexicallyDeclaredNames** of *FunctionStatementList* also occurs in the **VarDeclaredNames** of *FunctionStatementList*.
- It is a Syntax Error if **ContainsDuplicateLabels** of *FunctionStatementList* with argument « » is **true**.
- It is a Syntax Error if **ContainsUndefinedBreakTarget** of *FunctionStatementList* with argument « » is **true**.
- It is a Syntax Error if **ContainsUndefinedContinueTarget** of *FunctionStatementList* with arguments « » and « » is **true**.

15.2.2 Static Semantics: FunctionBodyContainsUseStrict

FunctionBody : *FunctionStatementList*

1. If the **Directive Prologue** of *FunctionBody* contains a **Use Strict Directive**, return **true**; otherwise, return **false**.

15.2.3 Runtime Semantics: EvaluateFunctionBody

With parameters *functionObject* and *argumentsList* (a [List](#)).

FunctionBody : *FunctionStatementList*

1. Perform ? [FunctionDeclarationInstantiation](#)(*functionObject*, *argumentsList*).
2. Return the result of evaluating *FunctionStatementList*.

15.2.4 Runtime Semantics: InstantiateOrdinaryFunctionObject

With parameter *scope*.

FunctionDeclaration : **function** *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Let *name* be [StringValue](#) of *BindingIdentifier*.
2. Let *sourceText* be the source text matched by *FunctionDeclaration*.
3. Let *F* be [OrdinaryFunctionCreate](#)(%[Function.prototype](#)%, *sourceText*, *FormalParameters*, *FunctionBody*, non-lexical-this, *scope*).
4. Perform [SetFunctionName](#)(*F*, *name*).
5. Perform [MakeConstructor](#)(*F*).
6. Return *F*.

FunctionDeclaration : **function** (*FormalParameters*) { *FunctionBody* }

1. Let *sourceText* be the source text matched by *FunctionDeclaration*.
2. Let *F* be [OrdinaryFunctionCreate](#)(%[Function.prototype](#)%, *sourceText*, *FormalParameters*, *FunctionBody*, non-lexical-this, *scope*).
3. Perform [SetFunctionName](#)(*F*, "default").
4. Perform [MakeConstructor](#)(*F*).
5. Return *F*.

NOTE

An anonymous *FunctionDeclaration* can only occur as part of an **export default** declaration, and its function code is therefore always [strict mode code](#).

15.2.5 Runtime Semantics: InstantiateOrdinaryFunctionExpression

With optional parameter *name*.

FunctionExpression : **function** (*FormalParameters*) { *FunctionBody* }

1. If *name* is not present, set *name* to "".
2. Let *scope* be the [LexicalEnvironment](#) of the [running execution context](#).
3. Let *sourceText* be the source text matched by *FunctionExpression*.
4. Let *closure* be [OrdinaryFunctionCreate](#)(%[Function.prototype](#)%, *sourceText*, *FormalParameters*, *FunctionBody*, non-lexical-this, *scope*).
5. Perform [SetFunctionName](#)(*closure*, *name*).
6. Perform [MakeConstructor](#)(*closure*).
7. Return *closure*.

FunctionExpression : **function** *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. **Assert**: *name* is not present.
2. Set *name* to **StringValue** of *BindingIdentifier*.
3. Let *scope* be the **running execution context**'s **LexicalEnvironment**.
4. Let *funcEnv* be **NewDeclarativeEnvironment**(*scope*).
5. Perform *funcEnv*.**CreateImmutableBinding**(*name*, **false**).
6. Let *sourceText* be the source text matched by *FunctionExpression*.
7. Let *closure* be **OrdinaryFunctionCreate**(%**Function.prototype**%, *sourceText*, *FormalParameters*, *FunctionBody*, **non-lexical-this**, *funcEnv*).
8. Perform **SetFunctionName**(*closure*, *name*).
9. Perform **MakeConstructor**(*closure*).
10. Perform *funcEnv*.**InitializeBinding**(*name*, *closure*).
11. Return *closure*.

NOTE The *BindingIdentifier* in a *FunctionExpression* can be referenced from inside the *FunctionExpression*'s *FunctionBody* to allow the function to call itself recursively. However, unlike in a *FunctionDeclaration*, the *BindingIdentifier* in a *FunctionExpression* cannot be referenced from and does not affect the scope enclosing the *FunctionExpression*.

15.2.6 Runtime Semantics: Evaluation

FunctionDeclaration : **function** *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return **NormalCompletion**(empty).

NOTE 1 An alternative semantics is provided in B.3.3.

FunctionDeclaration : **function** (*FormalParameters*) { *FunctionBody* }

1. Return **NormalCompletion**(empty).

FunctionExpression : **function** *BindingIdentifier*_{opt} (*FormalParameters*) { *FunctionBody* }

1. Return **InstantiateOrdinaryFunctionExpression** of *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**).

15.3 Arrow Function Definitions

Syntax

ArrowFunction [*In*, *Yield*, *Await*] :
 ArrowParameters [*?Yield*, *?Await*] [no *LineTerminator* here] => *ConciseBody* [*?In*]

ArrowParameters [*Yield*, *Await*] :
 BindingIdentifier [*?Yield*, *?Await*]
 CoverParenthesizedExpressionAndArrowParameterList [*?Yield*, *?Await*]

ConciseBody [*In*] :
 [lookahead ≠ {] *ExpressionBody* [*?In*, *~Await*]
 { *FunctionBody* [*~Yield*, *~Await*] }

ExpressionBody [*In*, *Await*] :
 AssignmentExpression [*?In*, *~Yield*, *?Await*]

Supplemental Syntax

When processing an instance of the production

ArrowParameters [*Yield*, *Await*] : *CoverParenthesizedExpressionAndArrowParameterList* [*?Yield*, *?Await*]

the interpretation of *CoverParenthesizedExpressionAndArrowParameterList* is refined using the following grammar:

ArrowFormalParameters [*Yield*, *Await*] :
 (*UniqueFormalParameters* [*?Yield*, *?Await*])

15.3.1 Static Semantics: Early Errors

ArrowFunction : *ArrowParameters* => *ConciseBody*

- It is a Syntax Error if *ArrowParameters* **Contains** *YieldExpression* is **true**.
- It is a Syntax Error if *ArrowParameters* **Contains** *AwaitExpression* is **true**.
- It is a Syntax Error if **ConciseBodyContainsUseStrict** of *ConciseBody* is **true** and **IsSimpleParameterList** of *ArrowParameters* is **false**.
- It is a Syntax Error if any element of the **BoundNames** of *ArrowParameters* also occurs in the **LexicallyDeclaredNames** of *ConciseBody*.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

- It is a Syntax Error if *CoverParenthesizedExpressionAndArrowParameterList* is not **covering** an *ArrowFormalParameters*.
- All **early error** rules for *ArrowFormalParameters* and its derived productions also apply to **CoveredFormalsList** of *CoverParenthesizedExpressionAndArrowParameterList*.

15.3.2 Static Semantics: ConciseBodyContainsUseStrict

ConciseBody : *ExpressionBody*

1. Return **false**.

ConciseBody : { *FunctionBody* }

1. Return **FunctionBodyContainsUseStrict** of *FunctionBody*.

15.3.3 Static Semantics: CoveredFormalsList

ArrowParameters : *BindingIdentifier*

1. Return this *ArrowParameters*.

CoverParenthesizedExpressionAndArrowParameterList :

(*Expression*)
(*Expression* ,)
()
(... *BindingIdentifier*)
(... *BindingPattern*)
(*Expression* , ... *BindingIdentifier*)
(*Expression* , ... *BindingPattern*)

1. Return the *ArrowFormalParameters* that is **covered** by *CoverParenthesizedExpressionAndArrowParameterList*.

15.3.4 Runtime Semantics: EvaluateConciseBody

With parameters *functionObject* and *argumentsList* (a *List*).

ConciseBody : *ExpressionBody*

1. Perform ? **FunctionDeclarationInstantiation**(*functionObject*, *argumentsList*).
2. Return the result of evaluating *ExpressionBody*.

15.3.5 Runtime Semantics: InstantiateArrowFunctionExpression

With optional parameter *name*.

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. If *name* is not present, set *name* to "".
2. Let *scope* be the *LexicalEnvironment* of the **running execution context**.
3. Let *sourceText* be the source text matched by *ArrowFunction*.
4. Let *parameters* be **CoveredFormalsList** of *ArrowParameters*.
5. Let *closure* be **OrdinaryFunctionCreate**(%**Function.prototype**%, *sourceText*, *parameters*, *ConciseBody*, **lexical-this**, *scope*).
6. Perform **SetFunctionName**(*closure*, *name*).
7. Return *closure*.

NOTE

An *ArrowFunction* does not define local bindings for **arguments**, **super**, **this**, or **new.target**. Any reference to **arguments**, **super**, **this**, or **new.target** within an *ArrowFunction* must resolve to a binding in a lexically enclosing environment. Typically this will be the *Function Environment* of an immediately enclosing function. Even though an *ArrowFunction* may contain references to **super**, the *function object* created in step 5 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*.

15.3.6 Runtime Semantics: Evaluation

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. Return [InstantiateArrowFunctionExpression](#) of *ArrowFunction*.

ExpressionBody : *AssignmentExpression*

1. Let *exprRef* be the result of evaluating *AssignmentExpression*.
2. Let *exprValue* be ? [GetValue](#)(*exprRef*).
3. Return [Completion](#) { [[Type]]: return, [[Value]]: *exprValue*, [[Target]]: empty }.

15.4 Method Definitions

Syntax

*MethodDefinition*_[Yield, Await] :

*PropertyName*_[?Yield, ?Await] (*UniqueFormalParameters*_[~Yield, ~Await]) {
 *FunctionBody*_[~Yield, ~Await] }

*GeneratorMethod*_[?Yield, ?Await]

*AsyncMethod*_[?Yield, ?Await]

*AsyncGeneratorMethod*_[?Yield, ?Await]

get *PropertyName*_[?Yield, ?Await] () { *FunctionBody*_[~Yield, ~Await] }

set *PropertyName*_[?Yield, ?Await] (*PropertySetParameterList*) { *FunctionBody*_[~Yield, ~Await] }

PropertySetParameterList :

*FormalParameter*_[~Yield, ~Await]

15.4.1 Static Semantics: Early Errors

MethodDefinition : *PropertyName* (*UniqueFormalParameters*) { *FunctionBody* }

- It is a Syntax Error if [FunctionBodyContainsUseStrict](#) of *FunctionBody* is **true** and [IsSimpleParameterList](#) of *UniqueFormalParameters* is **false**.
- It is a Syntax Error if any element of the [BoundNames](#) of *UniqueFormalParameters* 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 [FunctionBodyContainsUseStrict](#) of *FunctionBody* is **true** and [IsSimpleParameterList](#) of *PropertySetParameterList* is **false**.
- It is a Syntax Error if any element of the [BoundNames](#) of *PropertySetParameterList* also occurs in the [LexicallyDeclaredNames](#) of *FunctionBody*.

15.4.2 Static Semantics: HasDirectSuper

MethodDefinition : *PropertyName* (*UniqueFormalParameters*) { *FunctionBody* }

1. If *UniqueFormalParameters* [Contains SuperCall](#) is **true**, return **true**.

2. Return *FunctionBody* **Contains** *SuperCall*.

MethodDefinition : **get** *PropertyName* () { *FunctionBody* }

1. Return *FunctionBody* **Contains** *SuperCall*.

MethodDefinition : **set** *PropertyName* (*PropertySetParameterList*) { *FunctionBody* }

1. If *PropertySetParameterList* **Contains** *SuperCall* is **true**, return **true**.
2. Return *FunctionBody* **Contains** *SuperCall*.

GeneratorMethod : * *PropertyName* (*UniqueFormalParameters*) { *GeneratorBody* }

1. If *UniqueFormalParameters* **Contains** *SuperCall* is **true**, return **true**.
2. Return *GeneratorBody* **Contains** *SuperCall*.

AsyncGeneratorMethod : **async** * *PropertyName* (*UniqueFormalParameters*) { *AsyncGeneratorBody* }

1. If *UniqueFormalParameters* **Contains** *SuperCall* is **true**, return **true**.
2. Return *AsyncGeneratorBody* **Contains** *SuperCall*.

AsyncMethod : **async** *PropertyName* (*UniqueFormalParameters*) { *AsyncFunctionBody* }

1. If *UniqueFormalParameters* **Contains** *SuperCall* is **true**, return **true**.
2. Return *AsyncFunctionBody* **Contains** *SuperCall*.

15.4.3 Static Semantics: SpecialMethod

MethodDefinition : *PropertyName* (*UniqueFormalParameters*) { *FunctionBody* }

1. Return **false**.

MethodDefinition :

GeneratorMethod

AsyncMethod

AsyncGeneratorMethod

get *PropertyName* () { *FunctionBody* }

set *PropertyName* (*PropertySetParameterList*) { *FunctionBody* }

1. Return **true**.

15.4.4 Runtime Semantics: DefineMethod

With parameter *object* and optional parameter *functionPrototype*.

MethodDefinition : *PropertyName* (*UniqueFormalParameters*) { *FunctionBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.
2. **ReturnIfAbrupt**(*propKey*).
3. Let *scope* be the **running execution context**'s **LexicalEnvironment**.
4. If *functionPrototype* is present, then
 - a. Let *prototype* be *functionPrototype*.

5. Else,
 - a. Let *prototype* be `%Function.prototype%`.
6. Let *sourceText* be the source text matched by *MethodDefinition*.
7. Let *closure* be `OrdinaryFunctionCreate(prototype, sourceText, UniqueFormalParameters, FunctionBody, non-lexical-this, scope)`.
8. Perform `MakeMethod(closure, object)`.
9. Return the `Record` { `[[Key]]: propKey`, `[[Closure]]: closure` }.

15.4.5 Runtime Semantics: MethodDefinitionEvaluation

With parameters *object* and *enumerable*.

MethodDefinition : *PropertyName* (*UniqueFormalParameters*) { *FunctionBody* }

1. Let *methodDef* be ? `DefineMethod` of *MethodDefinition* with argument *object*.
2. Perform `SetFunctionName(methodDef.[[Closure]], methodDef.[[Key]])`.
3. Let *desc* be the `PropertyDescriptor` { `[[Value]]: methodDef.[[Closure]]`, `[[Writable]]: true`, `[[Enumerable]]: enumerable`, `[[Configurable]]: true` }.
4. Return ? `DefinePropertyOrThrow(object, methodDef.[[Key]], desc)`.

MethodDefinition : **get** *PropertyName* () { *FunctionBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.
2. `ReturnIfAbrupt(propKey)`.
3. Let *scope* be the `running execution context`'s `LexicalEnvironment`.
4. Let *sourceText* be the source text matched by *MethodDefinition*.
5. Let *formalParameterList* be an instance of the production `FormalParameters` : [empty] .
6. Let *closure* be `OrdinaryFunctionCreate(%Function.prototype%, sourceText, formalParameterList, FunctionBody, non-lexical-this, scope)`.
7. Perform `MakeMethod(closure, object)`.
8. Perform `SetFunctionName(closure, propKey, "get")`.
9. Let *desc* be the `PropertyDescriptor` { `[[Get]]: closure`, `[[Enumerable]]: enumerable`, `[[Configurable]]: true` }.
10. Return ? `DefinePropertyOrThrow(object, propKey, desc)`.

MethodDefinition : **set** *PropertyName* (*PropertySetParameterList*) { *FunctionBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.
2. `ReturnIfAbrupt(propKey)`.
3. Let *scope* be the `running execution context`'s `LexicalEnvironment`.
4. Let *sourceText* be the source text matched by *MethodDefinition*.
5. Let *closure* be `OrdinaryFunctionCreate(%Function.prototype%, sourceText, PropertySetParameterList, FunctionBody, non-lexical-this, scope)`.
6. Perform `MakeMethod(closure, object)`.
7. Perform `SetFunctionName(closure, propKey, "set")`.
8. Let *desc* be the `PropertyDescriptor` { `[[Set]]: closure`, `[[Enumerable]]: enumerable`, `[[Configurable]]: true` }.
9. Return ? `DefinePropertyOrThrow(object, propKey, desc)`.

GeneratorMethod : * *PropertyName* (*UniqueFormalParameters*) { *GeneratorBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.

2. `ReturnIfAbrupt(propKey)`.
3. Let *scope* be the `running execution context`'s `LexicalEnvironment`.
4. Let *sourceText* be the source text matched by `GeneratorMethod`.
5. Let *closure* be `OrdinaryFunctionCreate(%GeneratorFunction.prototype%, sourceText, UniqueFormalParameters, GeneratorBody, non-lexical-this, scope)`.
6. Perform `MakeMethod(closure, object)`.
7. Perform `SetFunctionName(closure, propKey)`.
8. Let *prototype* be `! OrdinaryObjectCreate(%GeneratorFunction.prototype.prototype%)`.
9. Perform `DefinePropertyOrThrow(closure, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.
10. Let *desc* be the `PropertyDescriptor` { `[[Value]]: closure, [[Writable]]: true, [[Enumerable]]: enumerable, [[Configurable]]: true` }.
11. Return `? DefinePropertyOrThrow(object, propKey, desc)`.

`AsyncGeneratorMethod` : **async** * *PropertyName* (*UniqueFormalParameters*) { *AsyncGeneratorBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.
2. `ReturnIfAbrupt(propKey)`.
3. Let *scope* be the `running execution context`'s `LexicalEnvironment`.
4. Let *sourceText* be the source text matched by `AsyncGeneratorMethod`.
5. Let *closure* be `! OrdinaryFunctionCreate(%AsyncGeneratorFunction.prototype%, sourceText, UniqueFormalParameters, AsyncGeneratorBody, non-lexical-this, scope)`.
6. Perform `! MakeMethod(closure, object)`.
7. Perform `! SetFunctionName(closure, propKey)`.
8. Let *prototype* be `! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.prototype%)`.
9. Perform `! DefinePropertyOrThrow(closure, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.
10. Let *desc* be `PropertyDescriptor` { `[[Value]]: closure, [[Writable]]: true, [[Enumerable]]: enumerable, [[Configurable]]: true` }.
11. Return `? DefinePropertyOrThrow(object, propKey, desc)`.

`AsyncMethod` : **async** *PropertyName* (*UniqueFormalParameters*) { *AsyncFunctionBody* }

1. Let *propKey* be the result of evaluating *PropertyName*.
2. `ReturnIfAbrupt(propKey)`.
3. Let *scope* be the `LexicalEnvironment` of the `running execution context`.
4. Let *sourceText* be the source text matched by `AsyncMethod`.
5. Let *closure* be `! OrdinaryFunctionCreate(%AsyncFunction.prototype%, sourceText, UniqueFormalParameters, AsyncFunctionBody, non-lexical-this, scope)`.
6. Perform `! MakeMethod(closure, object)`.
7. Perform `! SetFunctionName(closure, propKey)`.
8. Let *desc* be the `PropertyDescriptor` { `[[Value]]: closure, [[Writable]]: true, [[Enumerable]]: enumerable, [[Configurable]]: true` }.
9. Return `? DefinePropertyOrThrow(object, propKey, desc)`.

15.5 Generator Function Definitions

Syntax

*GeneratorMethod*_[Yield, Await] :

* *PropertyName*_[?Yield, ?Await] (*UniqueFormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

*GeneratorDeclaration*_[Yield, Await, Default] :

function * *BindingIdentifier*_[?Yield, ?Await] (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }
[+Default] function * (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

GeneratorExpression :

function * *BindingIdentifier*_[+Yield, ~Await] *opt* (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }
}

GeneratorBody :

*FunctionBody*_[+Yield, ~Await]

*YieldExpression*_[In, Await] :

yield

yield [no *LineTerminator* here] *AssignmentExpression*_[?In, +Yield, ?Await]

yield [no *LineTerminator* here] * *AssignmentExpression*_[?In, +Yield, ?Await]

NOTE 1 The syntactic context immediately following **yield** requires use of the *InputElementRegExpOrTemplateTail* lexical goal.

NOTE 2 *YieldExpression* cannot be used within the *FormalParameters* of a generator function because any expressions that are part of *FormalParameters* are evaluated before the resulting generator object is in a resumable state.

NOTE 3 [Abstract operations](#) relating to generator objects are defined in 27.5.3.

15.5.1 Static Semantics: Early Errors

GeneratorMethod : * *PropertyName* (*UniqueFormalParameters*) { *GeneratorBody* }

- It is a Syntax Error if [HasDirectSuper](#) of *GeneratorMethod* is **true**.
- It is a Syntax Error if *UniqueFormalParameters* [Contains](#) *YieldExpression* is **true**.
- It is a Syntax Error if [FunctionBodyContainsUseStrict](#) of *GeneratorBody* is **true** and [IsSimpleParameterList](#) of *UniqueFormalParameters* is **false**.
- It is a Syntax Error if any element of the [BoundNames](#) of *UniqueFormalParameters* also occurs in the [LexicallyDeclaredNames](#) of *GeneratorBody*.

GeneratorDeclaration :

function * *BindingIdentifier* (*FormalParameters*) { *GeneratorBody* }

function * (*FormalParameters*) { *GeneratorBody* }

GeneratorExpression :

function * *BindingIdentifier* *opt* (*FormalParameters*) { *GeneratorBody* }

- If the source code matching *FormalParameters* is [strict mode code](#), the Early Error rules for

UniqueFormalParameters : *FormalParameters* are applied.

- If *BindingIdentifier* is present and the source code matching *BindingIdentifier* is **strict mode code**, it is a Syntax Error if the **StringValue** of *BindingIdentifier* is "eval" or "arguments".
- It is a Syntax Error if **FunctionBodyContainsUseStrict** of *GeneratorBody* is **true** and **IsSimpleParameterList** of *FormalParameters* is **false**.
- It is a Syntax Error if any element of the **BoundNames** of *FormalParameters* also occurs in the **LexicallyDeclaredNames** of *GeneratorBody*.
- It is a Syntax Error if *FormalParameters* **Contains** *YieldExpression* is **true**.
- It is a Syntax Error if *FormalParameters* **Contains** *SuperProperty* is **true**.
- It is a Syntax Error if *GeneratorBody* **Contains** *SuperProperty* is **true**.
- It is a Syntax Error if *FormalParameters* **Contains** *SuperCall* is **true**.
- It is a Syntax Error if *GeneratorBody* **Contains** *SuperCall* is **true**.

15.5.2 Runtime Semantics: EvaluateGeneratorBody

With parameters *functionObject* and *argumentsList* (a **List**).

GeneratorBody : *FunctionBody*

1. Perform ? **FunctionDeclarationInstantiation**(*functionObject*, *argumentsList*).
2. Let *G* be ? **OrdinaryCreateFromConstructor**(*functionObject*, "%GeneratorFunction.prototype.prototype%", « [[GeneratorState]], [[GeneratorContext]], [[GeneratorBrand]] »).
3. Set *G*.[[GeneratorBrand]] to **empty**.
4. Perform **GeneratorStart**(*G*, *FunctionBody*).
5. Return **Completion** { [[Type]]: **return**, [[Value]]: *G*, [[Target]]: **empty** }.

15.5.3 Runtime Semantics: InstantiateGeneratorFunctionObject

With parameter *scope*.

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *GeneratorBody* }

1. Let *name* be **StringValue** of *BindingIdentifier*.
2. Let *sourceText* be the source text matched by *GeneratorDeclaration*.
3. Let *F* be **OrdinaryFunctionCreate**(%GeneratorFunction.prototype%, *sourceText*, *FormalParameters*, *GeneratorBody*, **non-lexical-this**, *scope*).
4. Perform **SetFunctionName**(*F*, *name*).
5. Let *prototype* be ! **OrdinaryObjectCreate**(%GeneratorFunction.prototype.prototype%).
6. Perform **DefinePropertyOrThrow**(*F*, "**prototype**", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
7. Return *F*.

GeneratorDeclaration : **function** * (*FormalParameters*) { *GeneratorBody* }

1. Let *sourceText* be the source text matched by *GeneratorDeclaration*.
2. Let *F* be **OrdinaryFunctionCreate**(%GeneratorFunction.prototype%, *sourceText*, *FormalParameters*, *GeneratorBody*, **non-lexical-this**, *scope*).
3. Perform **SetFunctionName**(*F*, "**default**").
4. Let *prototype* be ! **OrdinaryObjectCreate**(%GeneratorFunction.prototype.prototype%).

5. Perform `DefinePropertyOrThrow`(*F*, "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
6. Return *F*.

NOTE

An anonymous *GeneratorDeclaration* can only occur as part of an **export default** declaration, and its function code is therefore always **strict mode code**.

15.5.4 Runtime Semantics: InstantiateGeneratorFunctionExpression

With optional parameter *name*.

GeneratorExpression : **function** * (*FormalParameters*) { *GeneratorBody* }

1. If *name* is not present, set *name* to "".
2. Let *scope* be the LexicalEnvironment of the **running execution context**.
3. Let *sourceText* be the source text matched by *GeneratorExpression*.
4. Let *closure* be `OrdinaryFunctionCreate`(%*GeneratorFunction.prototype*%, *sourceText*, *FormalParameters*, *GeneratorBody*, non-lexical-this, *scope*).
5. Perform `SetFunctionName`(*closure*, *name*).
6. Let *prototype* be ! `OrdinaryObjectCreate`(%*GeneratorFunction.prototype.prototype*%).
7. Perform `DefinePropertyOrThrow`(*closure*, "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
8. Return *closure*.

GeneratorExpression : **function** * *BindingIdentifier* (*FormalParameters*) { *GeneratorBody* }

1. **Assert**: *name* is not present.
2. Set *name* to `StringValue` of *BindingIdentifier*.
3. Let *scope* be the **running execution context**'s LexicalEnvironment.
4. Let *funcEnv* be `NewDeclarativeEnvironment`(*scope*).
5. Perform *funcEnv*.`CreateImmutableBinding`(*name*, **false**).
6. Let *sourceText* be the source text matched by *GeneratorExpression*.
7. Let *closure* be `OrdinaryFunctionCreate`(%*GeneratorFunction.prototype*%, *sourceText*, *FormalParameters*, *GeneratorBody*, non-lexical-this, *funcEnv*).
8. Perform `SetFunctionName`(*closure*, *name*).
9. Let *prototype* be ! `OrdinaryObjectCreate`(%*GeneratorFunction.prototype.prototype*%).
10. Perform `DefinePropertyOrThrow`(*closure*, "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
11. Perform *funcEnv*.`InitializeBinding`(*name*, *closure*).
12. Return *closure*.

NOTE

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*.

15.5.5 Runtime Semantics: Evaluation

GeneratorExpression : **function** * *BindingIdentifier*_{opt} (*FormalParameters*) { *GeneratorBody* }

1. Return [InstantiateGeneratorFunctionExpression](#) of *GeneratorExpression*.

YieldExpression : **yield**

1. Return ? [Yield\(undefined\)](#).

YieldExpression : **yield** *AssignmentExpression*

1. Let *exprRef* be the result of evaluating *AssignmentExpression*.
2. Let *value* be ? [GetValue\(exprRef\)](#).
3. Return ? [Yield\(value\)](#).

YieldExpression : **yield** * *AssignmentExpression*

1. Let *generatorKind* be ! [GetGeneratorKind\(\)](#).
2. Let *exprRef* be the result of evaluating *AssignmentExpression*.
3. Let *value* be ? [GetValue\(exprRef\)](#).
4. Let *iteratorRecord* be ? [GetIterator\(value, generatorKind\)](#).
5. Let *iterator* be *iteratorRecord*.[[Iterator]].
6. Let *received* be [NormalCompletion\(undefined\)](#).
7. Repeat,
 - a. If *received*.[[Type]] is normal, then
 - i. Let *innerResult* be ? [Call\(iteratorRecord](#).[[NextMethod]], *iteratorRecord*.[[Iterator]], « *received*.[[Value]] »).
 - ii. If *generatorKind* is **async**, set *innerResult* to ? [Await\(innerResult\)](#).
 - iii. If [Type\(innerResult\)](#) is not Object, throw a **TypeError** exception.
 - iv. Let *done* be ? [IteratorComplete\(innerResult\)](#).
 - v. If *done* is **true**, then
 1. Return ? [IteratorValue\(innerResult\)](#).
 - vi. If *generatorKind* is **async**, set *received* to [AsyncGeneratorYield\(? IteratorValue\(innerResult\)\)](#).
 - vii. Else, set *received* to [GeneratorYield\(innerResult\)](#).
 - b. Else if *received*.[[Type]] is throw, then
 - i. Let *throw* be ? [GetMethod\(iterator, "throw"\)](#).
 - ii. If *throw* is not **undefined**, then
 1. Let *innerResult* be ? [Call\(throw, iterator, « received](#).[[Value]] »).
 2. If *generatorKind* is **async**, set *innerResult* to ? [Await\(innerResult\)](#).
 3. NOTE: Exceptions from the inner iterator **throw** method are propagated. Normal completions from an inner **throw** method are processed similarly to an inner **next**.
 4. If [Type\(innerResult\)](#) is not Object, throw a **TypeError** exception.
 5. Let *done* be ? [IteratorComplete\(innerResult\)](#).
 6. If *done* is **true**, then
 - a. Return ? [IteratorValue\(innerResult\)](#).
 7. If *generatorKind* is **async**, set *received* to [AsyncGeneratorYield\(? IteratorValue\(innerResult\)\)](#).
 8. Else, set *received* to [GeneratorYield\(innerResult\)](#).
 - iii. Else,
 1. NOTE: If *iterator* does not have a **throw** method, this throw is going to terminate the **yield*** loop. But first we need to give *iterator* a chance to clean up.

2. Let *closeCompletion* be **Completion** { [[Type]]: normal, [[Value]]: empty, [[Target]]: empty }.
 3. If *generatorKind* is **async**, perform ? **AsyncIteratorClose**(*iteratorRecord*, *closeCompletion*).
 4. Else, perform ? **IteratorClose**(*iteratorRecord*, *closeCompletion*).
 5. NOTE: The next step throws a **TypeError** to indicate that there was a **yield*** protocol violation: *iterator* does not have a **throw** method.
 6. Throw a **TypeError** exception.
- c. Else,
- i. **Assert**: *received*.[[Type]] is **return**.
 - ii. Let *return* be ? **GetMethod**(*iterator*, "return").
 - iii. If *return* is **undefined**, then
 1. If *generatorKind* is **async**, set *received*.[[Value]] to ? **Await**(*received*.[[Value]]).
 2. Return **Completion**(*received*).
 - iv. Let *innerReturnResult* be ? **Call**(*return*, *iterator*, « *received*.[[Value]] »).
 - v. If *generatorKind* is **async**, set *innerReturnResult* to ? **Await**(*innerReturnResult*).
 - vi. If **Type**(*innerReturnResult*) is not **Object**, throw a **TypeError** exception.
 - vii. Let *done* be ? **IteratorComplete**(*innerReturnResult*).
 - viii. If *done* is **true**, then
 1. Let *value* be ? **IteratorValue**(*innerReturnResult*).
 2. Return **Completion** { [[Type]]: **return**, [[Value]]: *value*, [[Target]]: empty }.
 - ix. If *generatorKind* is **async**, set *received* to **AsyncGeneratorYield**(? **IteratorValue**(*innerReturnResult*)).
 - x. Else, set *received* to **GeneratorYield**(*innerReturnResult*).

15.6 Async Generator Function Definitions

Syntax

*AsyncGeneratorMethod*_[Yield, Await] :

```
async [no LineTerminator here] * PropertyName[?Yield, ?Await] (
  UniqueFormalParameters[+Yield, +Await] ) { AsyncGeneratorBody }
```

*AsyncGeneratorDeclaration*_[Yield, Await, Default] :

```
async [no LineTerminator here] function * BindingIdentifier[?Yield, ?Await] (
  FormalParameters[+Yield, +Await] ) { AsyncGeneratorBody }
[+Default] async [no LineTerminator here] function * ( FormalParameters[+Yield, +Await] ) {
  AsyncGeneratorBody }
```

AsyncGeneratorExpression :

```
async [no LineTerminator here] function * BindingIdentifier[+Yield, +Await] opt (
  FormalParameters[+Yield, +Await] ) { AsyncGeneratorBody }
```

AsyncGeneratorBody :

```
FunctionBody[+Yield, +Await]
```

NOTE 1 *YieldExpression* and *AwaitExpression* cannot be used within the *FormalParameters* of an async generator function because any expressions that are part of *FormalParameters* are evaluated before the resulting async generator object is in a resumable state.

NOTE 2 [Abstract operations](#) relating to async generator objects are defined in [27.6.3](#).

15.6.1 Static Semantics: Early Errors

AsyncGeneratorMethod : **async** * *PropertyName* (*UniqueFormalParameters*) { *AsyncGeneratorBody* }

- It is a Syntax Error if [HasDirectSuper](#) of *AsyncGeneratorMethod* is **true**.
- It is a Syntax Error if *UniqueFormalParameters* [Contains](#) *YieldExpression* is **true**.
- It is a Syntax Error if *UniqueFormalParameters* [Contains](#) *AwaitExpression* is **true**.
- It is a Syntax Error if [FunctionBodyContainsUseStrict](#) of *AsyncGeneratorBody* is **true** and [IsSimpleParameterList](#) of *UniqueFormalParameters* is **false**.
- It is a Syntax Error if any element of the [BoundNames](#) of *UniqueFormalParameters* also occurs in the [LexicallyDeclaredNames](#) of *AsyncGeneratorBody*.

AsyncGeneratorDeclaration :

async function * *BindingIdentifier* (*FormalParameters*) { *AsyncGeneratorBody* }

async function * (*FormalParameters*) { *AsyncGeneratorBody* }

AsyncGeneratorExpression :

async function * *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncGeneratorBody* }

- If the source code matching *FormalParameters* is [strict mode code](#), the Early Error rules for *UniqueFormalParameters* : *FormalParameters* are applied.
- If *BindingIdentifier* is present and the source code matching *BindingIdentifier* is [strict mode code](#), it is a Syntax Error if the [StringValue](#) of *BindingIdentifier* is "eval" or "arguments".
- It is a Syntax Error if [FunctionBodyContainsUseStrict](#) of *AsyncGeneratorBody* is **true** and [IsSimpleParameterList](#) of *FormalParameters* is **false**.
- It is a Syntax Error if any element of the [BoundNames](#) of *FormalParameters* also occurs in the [LexicallyDeclaredNames](#) of *AsyncGeneratorBody*.
- It is a Syntax Error if *FormalParameters* [Contains](#) *YieldExpression* is **true**.
- It is a Syntax Error if *FormalParameters* [Contains](#) *AwaitExpression* is **true**.
- It is a Syntax Error if *FormalParameters* [Contains](#) *SuperProperty* is **true**.
- It is a Syntax Error if *AsyncGeneratorBody* [Contains](#) *SuperProperty* is **true**.
- It is a Syntax Error if *FormalParameters* [Contains](#) *SuperCall* is **true**.
- It is a Syntax Error if *AsyncGeneratorBody* [Contains](#) *SuperCall* is **true**.

15.6.2 Runtime Semantics: EvaluateAsyncGeneratorBody

With parameters *functionObject* and *argumentsList* (a [List](#)).

AsyncGeneratorBody : *FunctionBody*

1. Perform ? [FunctionDeclarationInstantiation](#)(*functionObject*, *argumentsList*).
2. Let *generator* be ? [OrdinaryCreateFromConstructor](#)(*functionObject*,

"%AsyncGeneratorFunction.prototype.prototype%", « [[AsyncGeneratorState]], [[AsyncGeneratorContext]], [[AsyncGeneratorQueue]], [[GeneratorBrand]] »).

3. Set *generator*.[[GeneratorBrand]] to empty.
4. Perform ! **AsyncGeneratorStart**(*generator*, *FunctionBody*).
5. Return **Completion** { [[Type]]: return, [[Value]]: *generator*, [[Target]]: empty }.

15.6.3 Runtime Semantics: InstantiateAsyncGeneratorFunctionObject

With parameter *scope*.

AsyncGeneratorDeclaration : **async function** * *BindingIdentifier* (*FormalParameters*) { *AsyncGeneratorBody* }

1. Let *name* be **StringValue** of *BindingIdentifier*.
2. Let *sourceText* be the source text matched by *AsyncGeneratorDeclaration*.
3. Let *F* be ! **OrdinaryFunctionCreate**(%**AsyncGeneratorFunction.prototype**%, *sourceText*, *FormalParameters*, *AsyncGeneratorBody*, non-lexical-this, *scope*).
4. Perform ! **SetFunctionName**(*F*, *name*).
5. Let *prototype* be ! **OrdinaryObjectCreate**(%**AsyncGeneratorFunction.prototype.prototype**%).
6. Perform ! **DefinePropertyOrThrow**(*F*, "**prototype**", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
7. Return *F*.

AsyncGeneratorDeclaration : **async function** * (*FormalParameters*) { *AsyncGeneratorBody* }

1. Let *sourceText* be the source text matched by *AsyncGeneratorDeclaration*.
2. Let *F* be **OrdinaryFunctionCreate**(%**AsyncGeneratorFunction.prototype**%, *sourceText*, *FormalParameters*, *AsyncGeneratorBody*, non-lexical-this, *scope*).
3. Perform **SetFunctionName**(*F*, "**default**").
4. Let *prototype* be ! **OrdinaryObjectCreate**(%**AsyncGeneratorFunction.prototype.prototype**%).
5. Perform **DefinePropertyOrThrow**(*F*, "**prototype**", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
6. Return *F*.

NOTE An anonymous *AsyncGeneratorDeclaration* can only occur as part of an **export default** declaration.

15.6.4 Runtime Semantics: InstantiateAsyncGeneratorFunctionExpression

With optional parameter *name*.

AsyncGeneratorExpression : **async function** * (*FormalParameters*) { *AsyncGeneratorBody* }

1. If *name* is not present, set *name* to "".
2. Let *scope* be the **LexicalEnvironment** of the **running execution context**.
3. Let *sourceText* be the source text matched by *AsyncGeneratorExpression*.
4. Let *closure* be ! **OrdinaryFunctionCreate**(%**AsyncGeneratorFunction.prototype**%, *sourceText*, *FormalParameters*, *AsyncGeneratorBody*, non-lexical-this, *scope*).
5. Perform **SetFunctionName**(*closure*, *name*).

6. Let *prototype* be ! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.prototype%).
7. Perform ! DefinePropertyOrThrow(*closure*, "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }).
8. Return *closure*.

AsyncGeneratorExpression : **async function** * *BindingIdentifier* (*FormalParameters*) { *AsyncGeneratorBody* }

1. Assert: *name* is not present.
2. Set *name* to StringValue of *BindingIdentifier*.
3. Let *scope* be the running execution context's LexicalEnvironment.
4. Let *funcEnv* be ! NewDeclarativeEnvironment(*scope*).
5. Perform ! *funcEnv*.CreateImmutableBinding(*name*, false).
6. Let *sourceText* be the source text matched by *AsyncGeneratorExpression*.
7. Let *closure* be ! OrdinaryFunctionCreate(%AsyncGeneratorFunction.prototype%, *sourceText*, *FormalParameters*, *AsyncGeneratorBody*, non-lexical-this, *funcEnv*).
8. Perform ! SetFunctionName(*closure*, *name*).
9. Let *prototype* be ! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.prototype%).
10. Perform ! DefinePropertyOrThrow(*closure*, "prototype", PropertyDescriptor { [[Value]]: *prototype*, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }).
11. Perform ! *funcEnv*.InitializeBinding(*name*, *closure*).
12. Return *closure*.

NOTE

The *BindingIdentifier* in an *AsyncGeneratorExpression* can be referenced from inside the *AsyncGeneratorExpression*'s *AsyncGeneratorBody* to allow the generator code to call itself recursively. However, unlike in an *AsyncGeneratorDeclaration*, the *BindingIdentifier* in an *AsyncGeneratorExpression* cannot be referenced from and does not affect the scope enclosing the *AsyncGeneratorExpression*.

15.6.5 Runtime Semantics: Evaluation

AsyncGeneratorExpression : **async function** * *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncGeneratorBody* }

1. Return InstantiateAsyncGeneratorFunctionExpression of *AsyncGeneratorExpression*.

15.7 Class Definitions

Syntax

```

ClassDeclaration[Yield, Await, Default] :
    class BindingIdentifier[?Yield, ?Await] ClassTail[?Yield, ?Await]
    [+Default] class ClassTail[?Yield, ?Await]

ClassExpression[Yield, Await] :
    class BindingIdentifier[?Yield, ?Await]opt ClassTail[?Yield, ?Await]

ClassTail[Yield, Await] :
    ClassHeritage[?Yield, ?Await]opt { ClassBody[?Yield, ?Await]opt }

```

```

ClassHeritage[Yield, Await] :
    extends LeftHandSideExpression[?Yield, ?Await]

ClassBody[Yield, Await] :
    ClassElementList[?Yield, ?Await]

ClassElementList[Yield, Await] :
    ClassElement[?Yield, ?Await]
    ClassElementList[?Yield, ?Await] ClassElement[?Yield, ?Await]

ClassElement[Yield, Await] :
    MethodDefinition[?Yield, ?Await]
    static MethodDefinition[?Yield, ?Await]
    ;

```

NOTE A class definition is always **strict mode code**.

15.7.1 Static Semantics: Early Errors

ClassTail : *ClassHeritage*_{opt} { *ClassBody* }

- It is a Syntax Error if *ClassHeritage* is not present and the following algorithm evaluates to **true**:
 1. Let *constructor* be *ConstructorMethod* of *ClassBody*.
 2. If *constructor* is empty, return **false**.
 3. Return *HasDirectSuper* of *constructor*.

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 not **"constructor"** and *HasDirectSuper* of *MethodDefinition* is **true**.
- 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 *HasDirectSuper* of *MethodDefinition* is **true**.
- It is a Syntax Error if *PropName* of *MethodDefinition* is **"prototype"**.

15.7.2 Static Semantics: ClassElementKind

ClassElement : *MethodDefinition*

1. If *PropName* of *MethodDefinition* is **"constructor"**, return *ConstructorMethod*.
2. Return *NonConstructorMethod*.

ClassElement : **static** *MethodDefinition*

1. Return **NonConstructorMethod**.

ClassElement : ;

1. Return **empty**.

15.7.3 Static Semantics: ConstructorMethod

ClassElementList : *ClassElement*

1. If **ClassElementKind** of *ClassElement* is **ConstructorMethod**, return *ClassElement*.
2. Return **empty**.

ClassElementList : *ClassElementList* *ClassElement*

1. Let *head* be **ConstructorMethod** of *ClassElementList*.
2. If *head* is not **empty**, return *head*.
3. If **ClassElementKind** of *ClassElement* is **ConstructorMethod**, return *ClassElement*.
4. Return **empty**.

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.

15.7.4 Static Semantics: IsStatic

ClassElement : *MethodDefinition*

1. Return **false**.

ClassElement : **static** *MethodDefinition*

1. Return **true**.

ClassElement : ;

1. Return **false**.

15.7.5 Static Semantics: NonConstructorMethodDefinitions

ClassElementList : *ClassElement*

1. If **ClassElementKind** of *ClassElement* is **NonConstructorMethod**, then
 - a. Return a **List** whose sole element is *ClassElement*.
2. Return a new empty **List**.

ClassElementList : *ClassElementList* *ClassElement*

1. Let *list* be **NonConstructorMethodDefinitions** of *ClassElementList*.
2. If **ClassElementKind** of *ClassElement* is **NonConstructorMethod**, then
 - a. Append *ClassElement* to the end of *list*.
3. Return *list*.

15.7.6 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** whose sole element is **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*.

15.7.7 Runtime Semantics: ClassDefinitionEvaluation

With parameters *classBinding* and *className*.

ClassTail : *ClassHeritage*_{opt} { *ClassBody*_{opt} }

1. Let *env* be the **LexicalEnvironment** of the **running execution context**.
2. Let *classScope* be **NewDeclarativeEnvironment**(*env*).
3. If *classBinding* is not **undefined**, then
 - a. Perform *classScope*.**CreateImmutableBinding**(*classBinding*, **true**).
4. If *ClassHeritage*_{opt} is not present, then
 - a. Let *protoParent* be **%Object.prototype%**.
 - b. Let *constructorParent* be **%Function.prototype%**.
5. Else,
 - a. Set the **running execution context**'s **LexicalEnvironment** to *classScope*.
 - b. Let *superclassRef* be the result of evaluating *ClassHeritage*.
 - c. Set the **running execution context**'s **LexicalEnvironment** to *env*.
 - d. Let *superclass* be **? GetValue**(*superclassRef*).
 - e. If *superclass* is **null**, then
 - i. Let *protoParent* be **null**.
 - ii. Let *constructorParent* be **%Function.prototype%**.
 - f. Else if **IsConstructor**(*superclass*) is **false**, throw a **TypeError** exception.
 - g. Else,
 - i. Let *protoParent* be **? Get**(*superclass*, **"prototype"**).
 - ii. If **Type**(*protoParent*) is neither **Object** nor **Null**, throw a **TypeError** exception.
 - iii. Let *constructorParent* be *superclass*.
6. Let *proto* be **! OrdinaryObjectCreate**(*protoParent*).
7. If *ClassBody*_{opt} is not present, let *constructor* be **empty**.
8. Else, let *constructor* be **ConstructorMethod** of *ClassBody*.
9. Set the **running execution context**'s **LexicalEnvironment** to *classScope*.
10. If *constructor* is **empty**, then
 - a. Let *steps* be the algorithm steps defined in **Default Constructor Functions**.
 - b. Let *F* be **! CreateBuiltinFunction**(*steps*, 0, *className*, « **[[ConstructorKind]]**, **[[SourceText]]** », **empty**, *constructorParent*).

11. Else,
 - a. Let *constructorInfo* be ! **DefineMethod** of *constructor* with arguments *proto* and *constructorParent*.
 - b. Let *F* be *constructorInfo*.[[Closure]].
 - c. Perform ! **MakeClassConstructor**(*F*).
 - d. Perform ! **SetFunctionName**(*F*, *className*).
12. Perform ! **MakeConstructor**(*F*, **false**, *proto*).
13. If *ClassHeritage*_{opt} is present, set *F*.[[ConstructorKind]] to **derived**.
14. Perform ! **CreateMethodProperty**(*proto*, "**constructor**", *F*).
15. If *ClassBody*_{opt} is not present, let *methods* be a new empty **List**.
16. Else, let *methods* be **NonConstructorMethodDefinitions** of *ClassBody*.
17. For each *ClassElement* *m* of *methods*, do
 - a. If **IsStatic** of *m* is **false**, then
 - i. Let *status* be **PropertyDefinitionEvaluation** of *m* with arguments *proto* and **false**.
 - b. Else,
 - i. Let *status* be **PropertyDefinitionEvaluation** of *m* with arguments *F* and **false**.
 - c. If *status* is an **abrupt completion**, then
 - i. Set the **running execution context**'s **LexicalEnvironment** to *env*.
 - ii. Return **Completion**(*status*).
18. Set the **running execution context**'s **LexicalEnvironment** to *env*.
19. If *classBinding* is not **undefined**, then
 - a. Perform *classScope*.**InitializeBinding**(*classBinding*, *F*).
20. Return *F*.

15.7.7.1 Default Constructor Functions

When a Default **Constructor** Function is called with zero or more arguments which form the rest parameter *...args*, the following steps are taken:

1. If **NewTarget** is **undefined**, throw a **TypeError** exception.
2. Let *F* be the **active function object**.
3. If *F*.[[ConstructorKind]] is **derived**, then
 - a. NOTE: This branch behaves similarly to **constructor(...args) { super(...args); }**. The most notable distinction is that while the aforementioned ECMAScript source text observably calls the **@@iterator** method on **%Array.prototype%**, a Default **Constructor** Function does not.
 - b. Let *func* be ! *F*.[[GetPrototypeOf]]().
 - c. If **IsConstructor**(*func*) is **false**, throw a **TypeError** exception.
 - d. Return ? **Construct**(*func*, *args*, **NewTarget**).
4. Else,
 - a. NOTE: This branch behaves similarly to **constructor() {}**.
 - b. Return ? **OrdinaryCreateFromConstructor**(**NewTarget**, "%**Object.prototype**%").

The "**length**" property of a default **constructor** function is +0_F.

15.7.8 Runtime Semantics: BindingClassDeclarationEvaluation

ClassDeclaration : **class** *BindingIdentifier* *ClassTail*

1. Let *className* be **StringValue** of *BindingIdentifier*.
2. Let *value* be ? **ClassDefinitionEvaluation** of *ClassTail* with arguments *className* and *className*.

3. Set *value*.[[SourceText]] to the source text matched by *ClassDeclaration*.
4. Let *env* be the [running execution context](#)'s `LexicalEnvironment`.
5. Perform ? `InitializeBoundName(className, value, env)`.
6. Return *value*.

ClassDeclaration : **class** *ClassTail*

1. Let *value* be ? `ClassDefinitionEvaluation` of *ClassTail* with arguments **undefined** and **"default"**.
2. Set *value*.[[SourceText]] to the source text matched by *ClassDeclaration*.
3. Return *value*.

NOTE

ClassDeclaration : **class** *ClassTail* only occurs as part of an *ExportDeclaration* and establishing its binding is handled as part of the evaluation action for that production. See [16.2.3.7](#).

15.7.9 Runtime Semantics: Evaluation

ClassDeclaration : **class** *BindingIdentifier* *ClassTail*

1. Perform ? `BindingClassDeclarationEvaluation` of this *ClassDeclaration*.
2. Return `NormalCompletion(empty)`.

NOTE

ClassDeclaration : **class** *ClassTail* only occurs as part of an *ExportDeclaration* and is never directly evaluated.

ClassExpression : **class** *ClassTail*

1. Let *value* be ? `ClassDefinitionEvaluation` of *ClassTail* with arguments **undefined** and **""**.
2. Set *value*.[[SourceText]] to the source text matched by *ClassExpression*.
3. Return *value*.

ClassExpression : **class** *BindingIdentifier* *ClassTail*

1. Let *className* be `StringValue` of *BindingIdentifier*.
2. Let *value* be ? `ClassDefinitionEvaluation` of *ClassTail* with arguments *className* and *className*.
3. Set *value*.[[SourceText]] to the source text matched by *ClassExpression*.
4. Return *value*.

15.8 Async Function Definitions

Syntax

```

AsyncFunctionDeclaration[Yield, Await, Default] :
  async [no LineTerminator here] function BindingIdentifier[?Yield, ?Await] (
    FormalParameters[~Yield, +Await] ) { AsyncFunctionBody }
[+Default] async [no LineTerminator here] function ( FormalParameters[~Yield, +Await] ) {
  AsyncFunctionBody }

```

AsyncFunctionExpression :

```
async [no LineTerminator here] function BindingIdentifier [~Yield, +Await] opt (
  FormalParameters [~Yield, +Await] ) { AsyncFunctionBody }
```

AsyncMethod [**Yield**, **Await**] :

```
async [no LineTerminator here] PropertyName [?Yield, ?Await] ( UniqueFormalParameters [~Yield, +Await]
) { AsyncFunctionBody }
```

AsyncFunctionBody :

```
FunctionBody [~Yield, +Await]
```

AwaitExpression [**Yield**] :

```
await UnaryExpression [?Yield, +Await]
```

NOTE 1

await is parsed as an *AwaitExpression* when the [**Await**] parameter is present. The [**Await**] parameter is present in the following contexts:

- In an *AsyncFunctionBody*.
- In the *FormalParameters* of an *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, or *AsyncGeneratorExpression*. *AwaitExpression* in this position is a Syntax error via [static semantics](#).

When *Module* is the syntactic [goal symbol](#) and the [**Await**] parameter is absent, **await** is parsed as a [keyword](#) and will be a Syntax error. When *Script* is the syntactic [goal symbol](#), **await** may be parsed as an identifier when the [**Await**] parameter is absent. This includes the following contexts:

- Anywhere outside of an *AsyncFunctionBody* or *FormalParameters* of an *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, *AsyncGeneratorDeclaration*, or *AsyncGeneratorExpression*.
- In the *BindingIdentifier* of a *FunctionExpression*, *GeneratorExpression*, or *AsyncGeneratorExpression*.

NOTE 2

Unlike *YieldExpression*, it is a Syntax Error to omit the operand of an *AwaitExpression*. You must await something.

15.8.1 Static Semantics: Early Errors

AsyncMethod : **async** PropertyName (UniqueFormalParameters) { AsyncFunctionBody }

- It is a Syntax Error if [FunctionBodyContainsUseStrict](#) of *AsyncFunctionBody* is **true** and [IsSimpleParameterList](#) of *UniqueFormalParameters* is **false**.
- It is a Syntax Error if [HasDirectSuper](#) of *AsyncMethod* is **true**.
- It is a Syntax Error if *UniqueFormalParameters* [Contains](#) *AwaitExpression* is **true**.
- It is a Syntax Error if any element of the [BoundNames](#) of *UniqueFormalParameters* also occurs in the [LexicallyDeclaredNames](#) of *AsyncFunctionBody*.

AsyncFunctionDeclaration :

async function *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

async function (*FormalParameters*) { *AsyncFunctionBody* }

AsyncFunctionExpression :

async function *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncFunctionBody* }

- It is a Syntax Error if **FunctionBodyContainsUseStrict** of *AsyncFunctionBody* is **true** and **IsSimpleParameterList** of *FormalParameters* is **false**.
- It is a Syntax Error if *FormalParameters* **Contains** *AwaitExpression* is **true**.
- If the source code matching *FormalParameters* is **strict mode code**, the Early Error rules for *UniqueFormalParameters* : *FormalParameters* are applied.
- If *BindingIdentifier* is present and the source code matching *BindingIdentifier* is **strict mode code**, it is a Syntax Error if the **StringValue** of *BindingIdentifier* is "eval" or "arguments".
- It is a Syntax Error if any element of the **BoundNames** of *FormalParameters* also occurs in the **LexicallyDeclaredNames** of *AsyncFunctionBody*.
- It is a Syntax Error if *FormalParameters* **Contains** *SuperProperty* is **true**.
- It is a Syntax Error if *AsyncFunctionBody* **Contains** *SuperProperty* is **true**.
- It is a Syntax Error if *FormalParameters* **Contains** *SuperCall* is **true**.
- It is a Syntax Error if *AsyncFunctionBody* **Contains** *SuperCall* is **true**.

15.8.2 Runtime Semantics: InstantiateAsyncFunctionObject

With parameter *scope*.

AsyncFunctionDeclaration : **async function** *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

1. Let *name* be **StringValue** of *BindingIdentifier*.
2. Let *sourceText* be the source text matched by *AsyncFunctionDeclaration*.
3. Let *F* be ! **OrdinaryFunctionCreate**(%*AsyncFunction.prototype*%, *sourceText*, *FormalParameters*, *AsyncFunctionBody*, non-lexical-this, *scope*).
4. Perform ! **SetFunctionName**(*F*, *name*).
5. Return *F*.

AsyncFunctionDeclaration : **async function** (*FormalParameters*) { *AsyncFunctionBody* }

1. Let *sourceText* be the source text matched by *AsyncFunctionDeclaration*.
2. Let *F* be ! **OrdinaryFunctionCreate**(%*AsyncFunction.prototype*%, *sourceText*, *FormalParameters*, *AsyncFunctionBody*, non-lexical-this, *scope*).
3. Perform ! **SetFunctionName**(*F*, "default").
4. Return *F*.

15.8.3 Runtime Semantics: InstantiateAsyncFunctionExpression

With optional parameter *name*.

AsyncFunctionExpression : **async function** (*FormalParameters*) { *AsyncFunctionBody* }

1. If *name* is not present, set *name* to "".
2. Let *scope* be the **LexicalEnvironment** of the **running execution context**.
3. Let *sourceText* be the source text matched by *AsyncFunctionExpression*.

4. Let *closure* be ! OrdinaryFunctionCreate(%AsyncFunction.prototype%, *sourceText*, *FormalParameters*, *AsyncFunctionBody*, non-lexical-this, *scope*).
5. Perform SetFunctionName(*closure*, *name*).
6. Return *closure*.

AsyncFunctionExpression : **async function** *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

1. **Assert**: *name* is not present.
2. Set *name* to StringValue of *BindingIdentifier*.
3. Let *scope* be the LexicalEnvironment of the running execution context.
4. Let *funcEnv* be ! NewDeclarativeEnvironment(*scope*).
5. Perform ! *funcEnv*.CreateImmutableBinding(*name*, **false**).
6. Let *sourceText* be the source text matched by *AsyncFunctionExpression*.
7. Let *closure* be ! OrdinaryFunctionCreate(%AsyncFunction.prototype%, *sourceText*, *FormalParameters*, *AsyncFunctionBody*, non-lexical-this, *funcEnv*).
8. Perform ! SetFunctionName(*closure*, *name*).
9. Perform ! *funcEnv*.InitializeBinding(*name*, *closure*).
10. Return *closure*.

NOTE The *BindingIdentifier* in an *AsyncFunctionExpression* can be referenced from inside the *AsyncFunctionExpression*'s *AsyncFunctionBody* to allow the function to call itself recursively. However, unlike in a *FunctionDeclaration*, the *BindingIdentifier* in a *AsyncFunctionExpression* cannot be referenced from and does not affect the scope enclosing the *AsyncFunctionExpression*.

15.8.4 Runtime Semantics: EvaluateAsyncFunctionBody

With parameters *functionObject* and *argumentsList* (a List).

AsyncFunctionBody : *FunctionBody*

1. Let *promiseCapability* be ! NewPromiseCapability(%Promise%).
2. Let *declResult* be FunctionDeclarationInstantiation(*functionObject*, *argumentsList*).
3. If *declResult* is not an abrupt completion, then
 - a. Perform ! AsyncFunctionStart(*promiseCapability*, *FunctionBody*).
4. Else,
 - a. Perform ! Call(*promiseCapability*.[[Reject]], **undefined**, « *declResult*.[[Value]] »).
5. Return Completion { [[Type]]: return, [[Value]]: *promiseCapability*.[[Promise]], [[Target]]: empty }.

15.8.5 Runtime Semantics: Evaluation

AsyncFunctionDeclaration : **async function** *BindingIdentifier* (*FormalParameters*) { *AsyncFunctionBody* }

1. Return NormalCompletion(empty).

AsyncFunctionDeclaration : **async function** (*FormalParameters*) { *AsyncFunctionBody* }

1. Return NormalCompletion(empty).

AsyncFunctionExpression :

async function *BindingIdentifier*_{opt} (*FormalParameters*) { *AsyncFunctionBody* }

1. Return `InstantiateAsyncFunctionExpression` of `AsyncFunctionExpression`.

AwaitExpression : **await** *UnaryExpression*

1. Let *exprRef* be the result of evaluating *UnaryExpression*.
2. Let *value* be ? `GetValue(exprRef)`.
3. Return ? `Await(value)`.

15.9 Async Arrow Function Definitions

Syntax

*AsyncArrowFunction*_[In, Yield, Await] :

async [no *LineTerminator* here] *AsyncArrowBindingIdentifier*_[?Yield] [no *LineTerminator* here] =>
*AsyncConciseBody*_[?In]

*CoverCallExpressionAndAsyncArrowHead*_[?Yield, ?Await] [no *LineTerminator* here] =>
*AsyncConciseBody*_[?In]

*AsyncConciseBody*_[In] :

[lookahead ≠ {] *ExpressionBody*_[?In, +Await]
 { *AsyncFunctionBody* }

*AsyncArrowBindingIdentifier*_[Yield] :

*BindingIdentifier*_[?Yield, +Await]

*CoverCallExpressionAndAsyncArrowHead*_[Yield, Await] :

*MemberExpression*_[?Yield, ?Await] *Arguments*_[?Yield, ?Await]

Supplemental Syntax

When processing an instance of the production

AsyncArrowFunction : *CoverCallExpressionAndAsyncArrowHead* => *AsyncConciseBody*

the interpretation of *CoverCallExpressionAndAsyncArrowHead* is refined using the following grammar:

AsyncArrowHead :

async [no *LineTerminator* here] *ArrowFormalParameters*_[~Yield, +Await]

15.9.1 Static Semantics: Early Errors

AsyncArrowFunction : **async** *AsyncArrowBindingIdentifier* => *AsyncConciseBody*

- It is a Syntax Error if any element of the `BoundNames` of *AsyncArrowBindingIdentifier* also occurs in the `LexicallyDeclaredNames` of *AsyncConciseBody*.

AsyncArrowFunction : *CoverCallExpressionAndAsyncArrowHead* => *AsyncConciseBody*

- It is a Syntax Error if *CoverCallExpressionAndAsyncArrowHead* `Contains YieldExpression` is **true**.
- It is a Syntax Error if *CoverCallExpressionAndAsyncArrowHead* `Contains AwaitExpression` is **true**.

- It is a Syntax Error if *CoverCallExpressionAndAsyncArrowHead* is not **covering** an *AsyncArrowHead*.
- It is a Syntax Error if any element of the **BoundNames** of *CoverCallExpressionAndAsyncArrowHead* also occurs in the **LexicallyDeclaredNames** of *AsyncConciseBody*.
- It is a Syntax Error if **AsyncConciseBodyContainsUseStrict** of *AsyncConciseBody* is **true** and **IsSimpleParameterList** of *CoverCallExpressionAndAsyncArrowHead* is **false**.
- All Early Error rules for *AsyncArrowHead* and its derived productions apply to **CoveredAsyncArrowHead** of *CoverCallExpressionAndAsyncArrowHead*.

15.9.2 Static Semantics: CoveredAsyncArrowHead

CoverCallExpressionAndAsyncArrowHead : *MemberExpression Arguments*

1. Return the *AsyncArrowHead* that is **covered** by *CoverCallExpressionAndAsyncArrowHead*.

15.9.3 Static Semantics: AsyncConciseBodyContainsUseStrict

AsyncConciseBody : *ExpressionBody*

1. Return **false**.

AsyncConciseBody : { *AsyncFunctionBody* }

1. Return **FunctionBodyContainsUseStrict** of *AsyncFunctionBody*.

15.9.4 Runtime Semantics: EvaluateAsyncConciseBody

With parameters *functionObject* and *argumentsList* (a **List**).

AsyncConciseBody : *ExpressionBody*

1. Let *promiseCapability* be ! **NewPromiseCapability**(%**Promise**%).
2. Let *declResult* be **FunctionDeclarationInstantiation**(*functionObject*, *argumentsList*).
3. If *declResult* is not an **abrupt completion**, then
 - a. Perform ! **AsyncFunctionStart**(*promiseCapability*, *ExpressionBody*).
4. Else,
 - a. Perform ! **Call**(*promiseCapability*.[[**Reject**]], **undefined**, « *declResult*.[[**Value**]] »).
5. Return **Completion** { [[**Type**]]: **return**, [[**Value**]]: *promiseCapability*.[[**Promise**]], [[**Target**]]: **empty** }.

15.9.5 Runtime Semantics: InstantiateAsyncArrowFunctionExpression

With optional parameter *name*.

AsyncArrowFunction : **async** *AsyncArrowBindingIdentifier* => *AsyncConciseBody*

1. If *name* is not present, set *name* to **""**.
2. Let *scope* be the **LexicalEnvironment** of the **running execution context**.
3. Let *sourceText* be the source text matched by *AsyncArrowFunction*.
4. Let *parameters* be *AsyncArrowBindingIdentifier*.
5. Let *closure* be ! **OrdinaryFunctionCreate**(%**AsyncFunction.prototype**%, *sourceText*, *parameters*, *AsyncConciseBody*, **lexical-this**, *scope*).
6. Perform **SetFunctionName**(*closure*, *name*).

7. Return *closure*.

AsyncArrowFunction : *CoverCallExpressionAndAsyncArrowHead* => *AsyncConciseBody*

1. If *name* is not present, set *name* to "".
2. Let *scope* be the LexicalEnvironment of the **running execution context**.
3. Let *sourceText* be the source text matched by *AsyncArrowFunction*.
4. Let *head* be **CoveredAsyncArrowHead** of *CoverCallExpressionAndAsyncArrowHead*.
5. Let *parameters* be the *ArrowFormalParameters* of *head*.
6. Let *closure* be ! **OrdinaryFunctionCreate**(%**AsyncFunction.prototype**%, *sourceText*, *parameters*, *AsyncConciseBody*, **lexical-this**, *scope*).
7. Perform **SetFunctionName**(*closure*, *name*).
8. Return *closure*.

15.9.6 Runtime Semantics: Evaluation

AsyncArrowFunction :

async *AsyncArrowBindingIdentifier* => *AsyncConciseBody*
CoverCallExpressionAndAsyncArrowHead => *AsyncConciseBody*

1. Return **InstantiateAsyncArrowFunctionExpression** of *AsyncArrowFunction*.

15.10 Tail Position Calls

15.10.1 Static Semantics: IsInTailPosition (*call*)

The abstract operation **IsInTailPosition** takes argument *call*. It performs the following steps when called:

1. **Assert**: *call* is a **Parse Node**.
2. If the source code matching *call* is **non-strict code**, return **false**.
3. If *call* is not contained within a *FunctionBody*, *ConciseBody*, or *AsyncConciseBody*, return **false**.
4. Let *body* be the *FunctionBody*, *ConciseBody*, or *AsyncConciseBody* that most closely contains *call*.
5. If *body* is the *FunctionBody* of a *GeneratorBody*, return **false**.
6. If *body* is the *FunctionBody* of an *AsyncFunctionBody*, return **false**.
7. If *body* is the *FunctionBody* of an *AsyncGeneratorBody*, return **false**.
8. If *body* is an *AsyncConciseBody*, return **false**.
9. Return the result of **HasCallInTailPosition** of *body* with argument *call*.

NOTE Tail Position calls are only defined in **strict mode code** because of a common non-standard language extension (see 10.2.4) that enables observation of the chain of caller contexts.

15.10.2 Static Semantics: HasCallInTailPosition

With parameter *call*.

NOTE

call is a [Parse Node](#) that represents a specific range of source text. When the following algorithms compare *call* to another [Parse Node](#), it is a test of whether they represent the same source text.

15.10.2.1 Statement Rules

StatementList : *StatementList* *StatementListItem*

1. Let *has* be [HasCallInTailPosition](#) of *StatementList* with argument *call*.
2. If *has* is **true**, return **true**.
3. Return [HasCallInTailPosition](#) of *StatementListItem* with argument *call*.

FunctionStatementList : [empty]

StatementListItem : *Declaration*

Statement :

VariableStatement

EmptyStatement

ExpressionStatement

ContinueStatement

BreakStatement

ThrowStatement

DebuggerStatement

Block : { }

ReturnStatement : **return** ;

LabelledItem : *FunctionDeclaration*

ForInOfStatement :

for (*LeftHandSideExpression* **of** *AssignmentExpression*) *Statement*

for (**var** *ForBinding* **of** *AssignmentExpression*) *Statement*

for (*ForDeclaration* **of** *AssignmentExpression*) *Statement*

CaseBlock : { }

1. Return **false**.

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *has* be [HasCallInTailPosition](#) of the first *Statement* with argument *call*.
2. If *has* is **true**, return **true**.
3. Return [HasCallInTailPosition](#) of the second *Statement* with argument *call*.

IfStatement : **if** (*Expression*) *Statement*

DoWhileStatement : **do** *Statement* **while** (*Expression*) ;

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

ForStatement :

for (*Expression*_{opt} ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

for (**var** *VariableDeclarationList* ; *Expression*_{opt} ; *Expression*_{opt}) *Statement*

for (*LexicalDeclaration* *Expression*_{opt} ; *Expression*_{opt}) *Statement*

ForInOfStatement :

for (*LeftHandSideExpression* **in** *Expression*) *Statement*

for (**var** *ForBinding* **in** *Expression*) *Statement*

for (*ForDeclaration in Expression*) *Statement*
for await (*LeftHandSideExpression of AssignmentExpression*) *Statement*
for await (**var** *ForBinding of AssignmentExpression*) *Statement*
for await (*ForDeclaration of AssignmentExpression*) *Statement*

WithStatement : **with** (*Expression*) *Statement*

1. Return [HasCallInTailPosition](#) of *Statement* with argument *call*.

LabelledStatement :

LabelIdentifier : *LabelledItem*

1. Return [HasCallInTailPosition](#) of *LabelledItem* with argument *call*.

ReturnStatement : **return** *Expression* ;

1. Return [HasCallInTailPosition](#) of *Expression* with argument *call*.

SwitchStatement : **switch** (*Expression*) *CaseBlock*

1. Return [HasCallInTailPosition](#) of *CaseBlock* with argument *call*.

CaseBlock : { *CaseClauses*_{opt} *DefaultClause* *CaseClauses*_{opt} }

1. Let *has* be **false**.
2. If the first *CaseClauses* is present, let *has* be [HasCallInTailPosition](#) of the first *CaseClauses* with argument *call*.
3. If *has* is **true**, return **true**.
4. Let *has* be [HasCallInTailPosition](#) of *DefaultClause* with argument *call*.
5. If *has* is **true**, return **true**.
6. If the second *CaseClauses* is present, let *has* be [HasCallInTailPosition](#) of the second *CaseClauses* with argument *call*.
7. Return *has*.

CaseClauses : *CaseClauses* *CaseClause*

1. Let *has* be [HasCallInTailPosition](#) of *CaseClauses* with argument *call*.
2. If *has* is **true**, return **true**.
3. Return [HasCallInTailPosition](#) of *CaseClause* with argument *call*.

CaseClause : **case** *Expression* : *StatementList*_{opt}

DefaultClause : **default** : *StatementList*_{opt}

1. If *StatementList* is present, return [HasCallInTailPosition](#) of *StatementList* with argument *call*.
2. Return **false**.

TryStatement : **try** *Block* *Catch*

1. Return [HasCallInTailPosition](#) of *Catch* with argument *call*.

TryStatement : **try** *Block* *Finally*

TryStatement : **try** *Block* *Catch* *Finally*

1. Return [HasCallInTailPosition](#) of *Finally* with argument *call*.

Catch : **catch** (*CatchParameter*) *Block*

1. Return [HasCallInTailPosition](#) of *Block* with argument *call*.

15.10.2.2 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. A function call cannot return a [Reference Record](#), so such a [GetValue](#) operation will always return the same value as the actual function call result.

AssignmentExpression :

YieldExpression

ArrowFunction

AsyncArrowFunction

LeftHandSideExpression = *AssignmentExpression*

LeftHandSideExpression *AssignmentOperator* *AssignmentExpression*

LeftHandSideExpression **&&=** *AssignmentExpression*

LeftHandSideExpression **||=** *AssignmentExpression*

LeftHandSideExpression **??=** *AssignmentExpression*

BitwiseANDExpression : *BitwiseANDExpression* **&** *EqualityExpression*

BitwiseXORExpression : *BitwiseXORExpression* **^** *BitwiseANDExpression*

BitwiseORExpression : *BitwiseORExpression* **|** *BitwiseXORExpression*

EqualityExpression :

EqualityExpression **==** *RelationalExpression*

EqualityExpression **!=** *RelationalExpression*

EqualityExpression **===** *RelationalExpression*

EqualityExpression **!==** *RelationalExpression*

RelationalExpression :

RelationalExpression **<** *ShiftExpression*

RelationalExpression **>** *ShiftExpression*

RelationalExpression **<=** *ShiftExpression*

RelationalExpression **>=** *ShiftExpression*

RelationalExpression **instanceof** *ShiftExpression*

RelationalExpression **in** *ShiftExpression*

ShiftExpression :

ShiftExpression **<<** *AdditiveExpression*

ShiftExpression **>>** *AdditiveExpression*

ShiftExpression **>>>** *AdditiveExpression*

AdditiveExpression :

AdditiveExpression **+** *MultiplicativeExpression*

AdditiveExpression **-** *MultiplicativeExpression*

MultiplicativeExpression :

MultiplicativeExpression *MultiplicativeOperator* *ExponentiationExpression*

ExponentiationExpression :

UpdateExpression ****** *ExponentiationExpression*

UpdateExpression :

LeftHandSideExpression ++

LeftHandSideExpression --

++ *UnaryExpression*

-- *UnaryExpression*

UnaryExpression :

delete *UnaryExpression*

void *UnaryExpression*

typeof *UnaryExpression*

+ *UnaryExpression*

- *UnaryExpression*

~ *UnaryExpression*

! *UnaryExpression*

AwaitExpression

CallExpression :

SuperCall

CallExpression [*Expression*]

CallExpression . *IdentifierName*

NewExpression : **new** *NewExpression*

MemberExpression :

MemberExpression [*Expression*]

MemberExpression . *IdentifierName*

SuperProperty

MetaProperty

new *MemberExpression* *Arguments*

PrimaryExpression :

this

IdentifierReference

Literal

ArrayLiteral

ObjectLiteral

FunctionExpression

ClassExpression

GeneratorExpression

AsyncFunctionExpression

AsyncGeneratorExpression

RegularExpressionLiteral

TemplateLiteral

1. Return **false**.

Expression :

AssignmentExpression

Expression , *AssignmentExpression*

1. Return **HasCallInTailPosition** of *AssignmentExpression* with argument *call*.

ConditionalExpression : *ShortCircuitExpression* ? *AssignmentExpression* : *AssignmentExpression*

1. Let *has* be **HasCallInTailPosition** of the first *AssignmentExpression* with argument *call*.
2. If *has* is **true**, return **true**.
3. Return **HasCallInTailPosition** of the second *AssignmentExpression* with argument *call*.

LogicalANDExpression : *LogicalANDExpression* && *BitwiseORExpression*

1. Return **HasCallInTailPosition** of *BitwiseORExpression* with argument *call*.

LogicalORExpression : *LogicalORExpression* || *LogicalANDExpression*

1. Return **HasCallInTailPosition** of *LogicalANDExpression* with argument *call*.

CoalesceExpression : *CoalesceExpressionHead* ?? *BitwiseORExpression*

1. Return **HasCallInTailPosition** of *BitwiseORExpression* with argument *call*.

CallExpression :

CoverCallExpressionAndAsyncArrowHead

CallExpression Arguments

CallExpression TemplateLiteral

1. If this *CallExpression* is *call*, return **true**.
2. Return **false**.

OptionalExpression :

MemberExpression OptionalChain

CallExpression OptionalChain

OptionalExpression OptionalChain

1. Return **HasCallInTailPosition** of *OptionalChain* with argument *call*.

OptionalChain :

? . [*Expression*]

? . *IdentifierName*

OptionalChain [*Expression*]

OptionalChain . *IdentifierName*

1. Return **false**.

OptionalChain :

? . *Arguments*

OptionalChain Arguments

1. If this *OptionalChain* is *call*, return **true**.
2. Return **false**.

MemberExpression :

MemberExpression TemplateLiteral

1. If this *MemberExpression* is *call*, return **true**.
2. Return **false**.

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be *CoveredParenthesizedExpression* of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return *HasCallInTailPosition* of *expr* with argument *call*.

ParenthesizedExpression :

(*Expression*)

1. Return *HasCallInTailPosition* of *Expression* with argument *call*.

15.10.3 PrepareForTailCall ()

The abstract operation *PrepareForTailCall* takes no arguments. It performs the following steps when called:

1. Let *leafContext* be the *running execution context*.
2. Suspend *leafContext*.
3. Pop *leafContext* from the *execution 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

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.

16 ECMAScript Language: Scripts and Modules

16.1 Scripts

Syntax

Script :

*ScriptBody*_{opt}

ScriptBody :

*StatementList*_[~Yield, ~Await, ~Return]

16.1.1 Static Semantics: Early Errors

Script : *ScriptBody*

- It is a Syntax Error if the [LexicallyDeclaredNames](#) of *ScriptBody* contains any duplicate entries.
- It is a Syntax Error if any element of the [LexicallyDeclaredNames](#) of *ScriptBody* also occurs in the [VarDeclaredNames](#) of *ScriptBody*.

ScriptBody : *StatementList*

- It is a Syntax Error if *StatementList* [Contains](#) **super** unless the source code containing **super** is eval code that is being processed by a [direct eval](#). Additional [early error](#) rules for **super** within [direct eval](#) are defined in [19.2.1.1](#).
- It is a Syntax Error if *StatementList* [Contains](#) *NewTarget* unless the source code containing *NewTarget* is eval code that is being processed by a [direct eval](#). Additional [early error](#) rules for *NewTarget* in [direct eval](#) are defined in [19.2.1.1](#).
- It is a Syntax Error if [ContainsDuplicateLabels](#) of *StatementList* with argument « » is **true**.
- It is a Syntax Error if [ContainsUndefinedBreakTarget](#) of *StatementList* with argument « » is **true**.
- It is a Syntax Error if [ContainsUndefinedContinueTarget](#) of *StatementList* with arguments « » and « » is **true**.

16.1.2 Static Semantics: IsStrict

Script : *ScriptBody*_{opt}

1. If *ScriptBody* is present and the [Directive Prologue](#) of *ScriptBody* contains a [Use Strict Directive](#), return **true**; otherwise, return **false**.

16.1.3 Runtime Semantics: Evaluation

Script : [empty]

1. Return [NormalCompletion\(undefined\)](#).

16.1.4 Script Records

A *Script Record* encapsulates information about a script being evaluated. Each script record contains the fields listed in [Table 39](#).

Table 39: [Script Record](#) Fields

Field Name	Value Type	Meaning
[[Realm]]	Realm Record undefined	The realm within which this script was created. undefined if not yet assigned.
[[Environment]]	Environment Record undefined	The Environment Record containing the top level bindings for this script. This field is set when the script is instantiated.
[[ECMAScriptCode]]	a Parse Node	The result of parsing the source text of this script using <i>Script</i> as the goal symbol .
[[HostDefined]]	Any, default value is empty .	Field reserved for use by host environments that need to associate additional information with a script.

16.1.5 ParseScript (*sourceText*, *realm*, *hostDefined*)

The abstract operation ParseScript takes arguments *sourceText*, *realm*, and *hostDefined*. It creates a [Script Record](#) based upon the result of parsing *sourceText* as a *Script*. It performs the following steps when called:

1. **Assert**: *sourceText* is an ECMAScript source text (see clause 11).
2. Let *body* be [ParseText](#)(*sourceText*, *Script*).
3. If *body* is a [List](#) of errors, return *body*.
4. Return [Script Record](#) { [\[\[Realm\]\]](#): *realm*, [\[\[Environment\]\]](#): **undefined**, [\[\[ECMAScriptCode\]\]](#): *body*, [\[\[HostDefined\]\]](#): *hostDefined* }.

NOTE An implementation may parse script source text and analyse it for Early Error conditions prior to evaluation of ParseScript for that script source text. However, the reporting of any errors must be deferred until the point where this specification actually performs ParseScript upon that source text.

16.1.6 ScriptEvaluation (*scriptRecord*)

The abstract operation ScriptEvaluation takes argument *scriptRecord*. It performs the following steps when called:

1. Let *globalEnv* be *scriptRecord*.[\[\[Realm\]\]](#).[\[\[GlobalEnv\]\]](#).
2. Let *scriptContext* be a new ECMAScript code [execution context](#).
3. Set the Function of *scriptContext* to **null**.
4. Set the [Realm](#) of *scriptContext* to *scriptRecord*.[\[\[Realm\]\]](#).
5. Set the ScriptOrModule of *scriptContext* to *scriptRecord*.
6. Set the VariableEnvironment of *scriptContext* to *globalEnv*.
7. Set the LexicalEnvironment of *scriptContext* to *globalEnv*.
8. Suspend the currently [running execution context](#).
9. Push *scriptContext* onto the [execution context stack](#); *scriptContext* is now the [running execution context](#).
10. Let *scriptBody* be *scriptRecord*.[\[\[ECMAScriptCode\]\]](#).
11. Let *result* be [GlobalDeclarationInstantiation](#)(*scriptBody*, *globalEnv*).
12. If *result*.[\[\[Type\]\]](#) is normal, then
 - a. Set *result* to the result of evaluating *scriptBody*.
13. If *result*.[\[\[Type\]\]](#) is normal and *result*.[\[\[Value\]\]](#) is empty, then
 - a. Set *result* to [NormalCompletion](#)(**undefined**).
14. Suspend *scriptContext* 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 [Completion](#)(*result*).

16.1.7 GlobalDeclarationInstantiation (*script*, *env*)

NOTE 1 When an [execution context](#) is established for evaluating scripts, declarations are instantiated in the current global environment. Each global binding declared in the code is instantiated.

The abstract operation GlobalDeclarationInstantiation takes arguments *script* (a [Parse Node](#) for *ScriptBody*) and *env* (an [Environment Record](#)). *script* is the *ScriptBody* for which the [execution context](#) is being established. *env* is the global

environment in which bindings are to be created. It performs the following steps when called:

1. **Assert:** *env* is a global [Environment Record](#).
 2. Let *lexNames* be the [LexicallyDeclaredNames](#) of *script*.
 3. Let *varNames* be the [VarDeclaredNames](#) of *script*.
 4. For each element *name* of *lexNames*, do
 - a. If *env*.[HasVarDeclaration](#)(*name*) is **true**, throw a **SyntaxError** exception.
 - b. If *env*.[HasLexicalDeclaration](#)(*name*) is **true**, throw a **SyntaxError** exception.
 - c. Let *hasRestrictedGlobal* be ? *env*.[HasRestrictedGlobalProperty](#)(*name*).
 - d. If *hasRestrictedGlobal* is **true**, throw a **SyntaxError** exception.
 5. For each element *name* of *varNames*, do
 - a. If *env*.[HasLexicalDeclaration](#)(*name*) is **true**, throw a **SyntaxError** exception.
 6. Let *varDeclarations* be the [VarScopedDeclarations](#) of *script*.
 7. Let *functionsToInitialize* be a new empty [List](#).
 8. Let *declaredFunctionNames* be a new empty [List](#).
 9. For each element *d* of *varDeclarations*, in reverse [List](#) order, do
 - a. If *d* is neither a [VariableDeclaration](#) nor a [ForBinding](#) nor a [BindingIdentifier](#), then
 - i. **Assert:** *d* is either a [FunctionDeclaration](#), a [GeneratorDeclaration](#), an [AsyncFunctionDeclaration](#), or an [AsyncGeneratorDeclaration](#).
 - ii. NOTE: If there are multiple function declarations 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 ? *env*.[CanDeclareGlobalFunction](#)(*fn*).
 2. If *fnDefinable* is **false**, throw a **TypeError** exception.
 3. Append *fn* to *declaredFunctionNames*.
 4. Insert *d* as the first element of *functionsToInitialize*.
 10. Let *declaredVarNames* be a new empty [List](#).
 11. For each element *d* of *varDeclarations*, do
 - a. If *d* is a [VariableDeclaration](#), a [ForBinding](#), or a [BindingIdentifier](#), then
 - i. For each String *vn* of the [BoundNames](#) of *d*, do
 1. If *vn* is not an element of *declaredFunctionNames*, then
 - a. Let *vnDefinable* be ? *env*.[CanDeclareGlobalVar](#)(*vn*).
 - b. If *vnDefinable* is **false**, throw a **TypeError** exception.
 - c. If *vn* is not an element of *declaredVarNames*, then
 - i. Append *vn* to *declaredVarNames*.
 12. NOTE: No abnormal terminations occur after this algorithm step 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. NOTE: Annex [B.3.3.2](#) adds additional steps at this point.
 14. Let *lexDeclarations* be the [LexicallyScopedDeclarations](#) of *script*.
 15. For each element *d* of *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. Perform ? *env*.[CreateImmutableBinding](#)(*dn*, **true**).
 - ii. Else,
 1. Perform ? *env*.[CreateMutableBinding](#)(*dn*, **false**).
16. For each [Parse Node](#) *f* of *functionsToInitialize*, do

- a. Let *fn* be the sole element of the `BoundNames` of *f*.
 - b. Let *fo* be `InstantiateFunctionObject` of *f* with argument *env*.
 - c. Perform `?env.CreateGlobalFunctionBinding(fn, fo, false)`.
17. For each String *vn* of `declaredVarNames`, do
- a. Perform `?env.CreateGlobalVarBinding(vn, false)`.
18. Return `NormalCompletion(empty)`.

NOTE 2

Early errors specified in 16.1.1 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 then 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.

16.2 Modules

Syntax

Module :

*ModuleBody*_{opt}

ModuleBody :

ModuleItemList

ModuleItemList :

ModuleItem

ModuleItemList *ModuleItem*

ModuleItem :

ImportDeclaration

ExportDeclaration

StatementListItem [*~Yield*, *~Await*, *~Return*]

16.2.1 Module Semantics

16.2.1.1 Static Semantics: Early Errors

ModuleBody : *ModuleItemList*

- It is a Syntax Error if the `LexicallyDeclaredNames` of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if any element of the `LexicallyDeclaredNames` of *ModuleItemList* also occurs in the `VarDeclaredNames` of *ModuleItemList*.

- It is a Syntax Error if the [ExportedNames](#) of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if any element of the [ExportedBindings](#) of *ModuleItemList* does not also occur in either the [VarDeclaredNames](#) of *ModuleItemList*, or the [LexicallyDeclaredNames](#) of *ModuleItemList*.
- It is a Syntax Error if *ModuleItemList* **Contains `super`**.
- It is a Syntax Error if *ModuleItemList* **Contains `NewTarget`**.
- It is a Syntax Error if [ContainsDuplicateLabels](#) of *ModuleItemList* with argument « » is **true**.
- It is a Syntax Error if [ContainsUndefinedBreakTarget](#) of *ModuleItemList* with argument « » is **true**.
- It is a Syntax Error if [ContainsUndefinedContinueTarget](#) of *ModuleItemList* with arguments « » and « » is **true**.

NOTE The duplicate [ExportedNames](#) rule implies that multiple **export default** *ExportDeclaration* items within a *ModuleBody* is a Syntax Error. 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.

16.2.1.2 Static Semantics: ImportedLocalNames (*importEntries*)

The abstract operation ImportedLocalNames takes argument *importEntries* (a [List](#) of ImportEntry Records (see [Table 45](#))). It creates a [List](#) of all of the local name bindings defined by *importEntries*. It performs the following steps when called:

1. Let *localNames* be a new empty [List](#).
2. For each [ImportEntry Record](#) *i* of *importEntries*, do
 - a. Append *i*.[[LocalName]] to *localNames*.
3. Return *localNames*.

16.2.1.3 Static Semantics: ModuleRequests

Module : [empty]

1. Return a new empty [List](#).

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](#).

ImportDeclaration : **import** *ImportClause* *FromClause* ;

1. Return [ModuleRequests](#) of *FromClause*.

ModuleSpecifier : *StringLiteral*

1. Return a [List](#) whose sole element is the [SV](#) of *StringLiteral*.

ExportDeclaration : **export** *ExportFromClause FromClause* ;

1. Return the [ModuleRequests](#) of *FromClause*.

ExportDeclaration :

```
export NamedExports ;
export VariableStatement
export Declaration
export default HoistableDeclaration
export default ClassDeclaration
export default AssignmentExpression ;
```

1. Return a new empty [List](#).

16.2.1.4 Abstract Module Records

A *Module Record* encapsulates structural information about the imports and exports of a single module. This information is used to link the imports and exports of sets of connected modules. A Module Record includes four fields that are only used when evaluating a module.

For specification purposes Module Record values are values of the [Record](#) specification type and can be thought of as existing in a simple object-oriented hierarchy where Module Record is an abstract class with both abstract and concrete subclasses. This specification defines the abstract subclass named [Cyclic Module Record](#) and its concrete subclass named [Source Text Module Record](#). Other specifications and implementations may define additional Module Record subclasses corresponding to alternative module definition facilities that they defined.

Module Record defines the fields listed in [Table 40](#). All Module Definition subclasses include at least those fields. Module Record also defines the abstract method list in [Table 41](#). All Module definition subclasses must provide concrete implementations of these abstract methods.

Table 40: [Module Record](#) Fields

Field Name	Value Type	Meaning
[[Realm]]	Realm Record undefined	The Realm within which this module was created. undefined if not yet assigned.
[[Environment]]	module Environment Record undefined	The Environment Record containing the top level bindings for this module. This field is set when the module is linked.
[[Namespace]]	Object undefined	The Module Namespace Object (28.3) if one has been created for this module. Otherwise undefined .
[[HostDefined]]	Any, default value is undefined .	Field reserved for use by host environments that need to associate additional information with a module.

Table 41: Abstract Methods of Module Records

Method	Purpose
GetExportedNames(<i>exportStarSet</i>)	Return a list of all names that are either directly or indirectly exported from this module.
ResolveExport(<i>exportName</i> [, <i>resolveSet</i>])	<p>Return the binding of a name exported by this module. Bindings are represented by a <i>ResolvedBinding Record</i>, of the form { [[Module]]: Module Record, [[BindingName]]: String }. If the export is a Module Namespace Object without a direct binding in any module, [[BindingName]] will be set to "*namespace". Return null if the name cannot be resolved, or "ambiguous" if multiple bindings were found.</p> <p>Each time this operation is called with a specific <i>exportName</i>, <i>resolveSet</i> pair as arguments it must return the same result if it completes normally.</p>
Link()	Prepare the module for evaluation by transitively resolving all module dependencies and creating a module Environment Record .
Evaluate()	<p>If this module has already been evaluated successfully, return undefined; if it has already been evaluated unsuccessfully, throw the exception that was produced. Otherwise, transitively evaluate all module dependencies of this module and then evaluate this module.</p> <p>Link must have completed successfully prior to invoking this method.</p>

16.2.1.5 Cyclic Module Records

A *Cyclic Module Record* is used to represent information about a module that can participate in dependency cycles with other modules that are subclasses of the [Cyclic Module Record](#) type. Module Records that are not subclasses of the [Cyclic Module Record](#) type must not participate in dependency cycles with Source Text Module Records.

In addition to the fields defined in [Table 40](#) Cyclic Module Records have the additional fields listed in [Table 42](#)

Table 42: Additional Fields of Cyclic Module Records

Field Name	Value Type	Meaning
[[Status]]	unlinked linking linked evaluating evaluated	Initially unlinked. Transitions to linking, linked, evaluating, evaluated (in that order) as the module progresses throughout its lifecycle.
[[EvaluationError]]	An abrupt completion undefined	A completion of type throw representing the exception that occurred during evaluation. undefined if no exception occurred or if [[Status]] is not evaluated.
[[DFSIndex]]	Integer undefined	Auxiliary field used during Link and Evaluate only. If [[Status]] is linking or evaluating, this non-negative number records the point at which the module was first visited during the ongoing depth-first traversal of the dependency graph.
[[DFSAncestorIndex]]	Integer undefined	Auxiliary field used during Link and Evaluate only. If [[Status]] is linking or evaluating, this is either the module's own [[DFSIndex]] or that of an "earlier" module in the same strongly connected component.
[[RequestedModules]]	List of String	A List of all the <i>ModuleSpecifier</i> strings used by the module represented by this record to request the importation of a module. The List is source code occurrence ordered.

In addition to the methods defined in Table 41 Cyclic Module Records have the additional methods listed in Table 43

Table 43: Additional Abstract Methods of Cyclic Module Records

Method	Purpose
<code>InitializeEnvironment()</code>	Initialize the Environment Record of the module, including resolving all imported bindings, and create the module's execution context .
<code>ExecuteModule()</code>	Evaluate the module's code within its execution context .

16.2.1.5.1 Link () Concrete Method

The Link concrete method of a **Cyclic Module Record** *module* takes no arguments. On success, Link transitions this module's [[Status]] from unlinked to linked. On failure, an exception is thrown and this module's [[Status]] remains unlinked. (Most of the work is done by the auxiliary function **InnerModuleLinking**.) It performs the following steps when called:

1. **Assert**: *module*.[[Status]] is not linking or evaluating.
2. Let *stack* be a new empty **List**.
3. Let *result* be **InnerModuleLinking**(*module*, *stack*, 0).
4. If *result* is an **abrupt completion**, then
 - a. For each **Cyclic Module Record** *m* of *stack*, do

- i. **Assert:** $m.[[Status]]$ is linking.
 - ii. Set $m.[[Status]]$ to **unlinked**.
 - iii. Set $m.[[Environment]]$ to **undefined**.
 - iv. Set $m.[[DFSIndex]]$ to **undefined**.
 - v. Set $m.[[DFSAncestorIndex]]$ to **undefined**.
- b. **Assert:** $module.[[Status]]$ is **unlinked**.
 - c. Return *result*.
5. **Assert:** $module.[[Status]]$ is **linked** or **evaluated**.
 6. **Assert:** *stack* is empty.
 7. Return **undefined**.

16.2.1.5.1.1 InnerModuleLinking (*module*, *stack*, *index*)

The abstract operation InnerModuleLinking takes arguments *module* (a **Cyclic Module Record**), *stack*, and *index* (a non-negative **integer**). It is used by Link to perform the actual linking process for *module*, as well as recursively on all other modules in the dependency graph. The *stack* and *index* parameters, as well as a module's $[[DFSIndex]]$ and $[[DFSAncestorIndex]]$ fields, keep track of the depth-first search (DFS) traversal. In particular, $[[DFSAncestorIndex]]$ is used to discover strongly connected components (SCCs), such that all modules in an SCC transition to linked together. It performs the following steps when called:

1. If *module* is not a **Cyclic Module Record**, then
 - a. Perform ? *module*.Link().
 - b. Return *index*.
2. If $module.[[Status]]$ is **linking**, **linked**, or **evaluated**, then
 - a. Return *index*.
3. **Assert:** $module.[[Status]]$ is **unlinked**.
4. Set $module.[[Status]]$ to **linking**.
5. Set $module.[[DFSIndex]]$ to *index*.
6. Set $module.[[DFSAncestorIndex]]$ to *index*.
7. Set *index* to *index* + 1.
8. Append *module* to *stack*.
9. For each String *required* of $module.[[RequestedModules]]$, do
 - a. Let *requiredModule* be ? **HostResolveImportedModule**(*module*, *required*).
 - b. Set *index* to ? **InnerModuleLinking**(*requiredModule*, *stack*, *index*).
 - c. If *requiredModule* is a **Cyclic Module Record**, then
 - i. **Assert:** $requiredModule.[[Status]]$ is either **linking**, **linked**, or **evaluated**.
 - ii. **Assert:** $requiredModule.[[Status]]$ is **linking** if and only if *requiredModule* is in *stack*.
 - iii. If $requiredModule.[[Status]]$ is **linking**, then
 1. Set $module.[[DFSAncestorIndex]]$ to **min**($module.[[DFSAncestorIndex]]$, $requiredModule.[[DFSAncestorIndex]]$).
10. Perform ? *module*.InitializeEnvironment().
11. **Assert:** *module* occurs exactly once in *stack*.
12. **Assert:** $module.[[DFSAncestorIndex]] \leq module.[[DFSIndex]]$.
13. If $module.[[DFSAncestorIndex]] = module.[[DFSIndex]]$, then
 - a. Let *done* be **false**.
 - b. Repeat, while *done* is **false**,
 - i. Let *requiredModule* be the last element in *stack*.
 - ii. Remove the last element of *stack*.

- iii. **Assert:** *requiredModule* is a **Cyclic Module Record**.
 - iv. Set *requiredModule*.[[Status]] to **linked**.
 - v. If *requiredModule* and *module* are the same **Module Record**, set *done* to **true**.
14. Return *index*.

16.2.1.5.2 Evaluate () Concrete Method

The Evaluate concrete method of a **Cyclic Module Record** *module* takes no arguments. Evaluate transitions this module's [[Status]] from **linked** to **evaluated**. If execution results in an exception, that exception is recorded in the [[EvaluationError]] field and rethrown by future invocations of Evaluate. (Most of the work is done by the auxiliary function **InnerModuleEvaluation**.) It performs the following steps when called:

1. **Assert:** This call to Evaluate is not happening at the same time as another call to Evaluate within the **surrounding agent**.
2. **Assert:** *module*.[[Status]] is **linked** or **evaluated**.
3. Let *stack* be a new empty **List**.
4. Let *result* be **InnerModuleEvaluation**(*module*, *stack*, 0).
5. If *result* is an **abrupt completion**, then
 - a. For each **Cyclic Module Record** *m* of *stack*, do
 - i. **Assert:** *m*.[[Status]] is **evaluating**.
 - ii. Set *m*.[[Status]] to **evaluated**.
 - iii. Set *m*.[[EvaluationError]] to *result*.
 - b. **Assert:** *module*.[[Status]] is **evaluated** and *module*.[[EvaluationError]] is *result*.
 - c. Return *result*.
6. **Assert:** *module*.[[Status]] is **evaluated** and *module*.[[EvaluationError]] is **undefined**.
7. **Assert:** *stack* is empty.
8. Return **undefined**.

16.2.1.5.2.1 InnerModuleEvaluation (*module*, *stack*, *index*)

The abstract operation **InnerModuleEvaluation** takes arguments *module* (a **Module Record**), *stack*, and *index* (a non-negative **integer**). It is used by Evaluate to perform the actual evaluation process for *module*, as well as recursively on all other modules in the dependency graph. The *stack* and *index* parameters, as well as *module*'s [[DFSIndex]] and [[DFSAncestorIndex]] fields, are used the same way as in **InnerModuleLinking**. It performs the following steps when called:

1. If *module* is not a **Cyclic Module Record**, then
 - a. Perform ? *module*.Evaluate().
 - b. Return *index*.
2. If *module*.[[Status]] is **evaluated**, then
 - a. If *module*.[[EvaluationError]] is **undefined**, return *index*.
 - b. Otherwise, return *module*.[[EvaluationError]].
3. If *module*.[[Status]] is **evaluating**, return *index*.
4. **Assert:** *module*.[[Status]] is **linked**.
5. Set *module*.[[Status]] to **evaluating**.
6. Set *module*.[[DFSIndex]] to *index*.
7. Set *module*.[[DFSAncestorIndex]] to *index*.
8. Set *index* to *index* + 1.
9. Append *module* to *stack*.

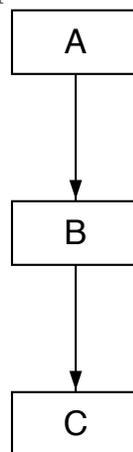
10. For each String *required* of *module*.[[RequestedModules]], do
 - a. Let *requiredModule* be ! `HostResolveImportedModule(module, required)`.
 - b. NOTE: Link must be completed successfully prior to invoking this method, so every requested module is guaranteed to resolve successfully.
 - c. Set *index* to ? `InnerModuleEvaluation(requiredModule, stack, index)`.
 - d. If *requiredModule* is a `Cyclic Module Record`, then
 - i. **Assert:** *requiredModule*.[[Status]] is either evaluating or evaluated.
 - ii. **Assert:** *requiredModule*.[[Status]] is evaluating if and only if *requiredModule* is in *stack*.
 - iii. If *requiredModule*.[[Status]] is evaluating, then
 1. Set *module*.[[DFSAncestorIndex]] to `min(module. [[DFSAncestorIndex]], requiredModule. [[DFSAncestorIndex]])`.
11. Perform ? `module.ExecuteModule()`.
12. **Assert:** *module* occurs exactly once in *stack*.
13. **Assert:** *module*.[[DFSAncestorIndex]] ≤ *module*.[[DFSIndex]].
14. If *module*.[[DFSAncestorIndex]] = *module*.[[DFSIndex]], then
 - a. Let *done* be **false**.
 - b. Repeat, while *done* is **false**,
 - i. Let *requiredModule* be the last element in *stack*.
 - ii. Remove the last element of *stack*.
 - iii. **Assert:** *requiredModule* is a `Cyclic Module Record`.
 - iv. Set *requiredModule*.[[Status]] to **evaluated**.
 - v. If *requiredModule* and *module* are the same `Module Record`, set *done* to **true**.
15. Return *index*.

16.2.1.5.3 Example Cyclic Module Record Graphs

This non-normative section gives a series of examples of the linking and evaluation of a few common module graphs, with a specific focus on how errors can occur.

First consider the following simple module graph:

Figure 2: A simple module graph



Let's first assume that there are no error conditions. When a `host` first calls `A.Link()`, this will complete successfully by assumption, and recursively link modules `B` and `C` as well, such that `A. [[Status]] = B. [[Status]] = C. [[Status]] = linked`. This preparatory step can be performed at any time. Later, when the `host` is ready to incur any possible side effects of

the modules, it can call `A.Evaluate()`, which will complete successfully (again by assumption), recursively having evaluated first `C` and then `B`. Each module's `[[Status]]` at this point will be `evaluated`.

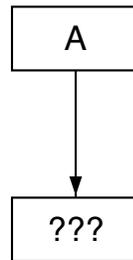
Consider then cases involving linking errors. If `InnerModuleLinking` of `C` succeeds but, thereafter, fails for `B`, for example because it imports something that `C` does not provide, then the original `A.Link()` will fail, and both `A` and `B`'s `[[Status]]` remain `unlinked`. `C`'s `[[Status]]` has become `linked`, though.

Finally, consider a case involving evaluation errors. If `InnerModuleEvaluation` of `C` succeeds but, thereafter, fails for `B`, for example because `B` contains code that throws an exception, then the original `A.Evaluate()` will fail. The resulting exception will be recorded in both `A` and `B`'s `[[EvaluationError]]` fields, and their `[[Status]]` will become `evaluated`. `C` will also become `evaluated` but, in contrast to `A` and `B`, will remain without an `[[EvaluationError]]`, as it successfully completed evaluation. Storing the exception ensures that any time a `host` tries to reuse `A` or `B` by calling their `Evaluate()` method, it will encounter the same exception. (Hosts are not required to reuse Cyclic Module Records; similarly, hosts are not required to expose the exception objects thrown by these methods. However, the specification enables such uses.)

The difference here between linking and evaluation errors is due to how evaluation must be only performed once, as it can cause side effects; it is thus important to remember whether evaluation has already been performed, even if unsuccessfully. (In the error case, it makes sense to also remember the exception because otherwise subsequent `Evaluate()` calls would have to synthesize a new one.) Linking, on the other hand, is side-effect-free, and thus even if it fails, it can be retried at a later time with no issues.

Now consider a different type of error condition:

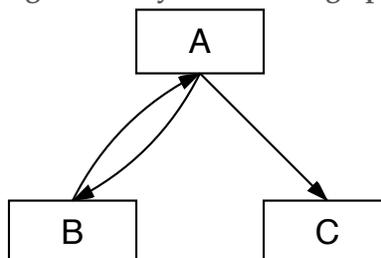
Figure 3: A module graph with an unresolvable module



In this scenario, module `A` declares a dependency on some other module, but no `Module Record` exists for that module, i.e. `HostResolveImportedModule` throws an exception when asked for it. This could occur for a variety of reasons, such as the corresponding resource not existing, or the resource existing but `ParseModule` throwing an exception when trying to parse the resulting source text. Hosts can choose to expose the cause of failure via the exception they throw from `HostResolveImportedModule`. In any case, this exception causes a linking failure, which as before results in `A`'s `[[Status]]` remaining `unlinked`.

Lastly, consider a module graph with a cycle:

Figure 4: A cyclic module graph



Here we assume that the entry point is module *A*, so that the `host` proceeds by calling `A.Link()`, which performs `InnerModuleLinking` on *A*. This in turn calls `InnerModuleLinking` on *B*. Because of the cycle, this again triggers `InnerModuleLinking` on *A*, but at this point it is a no-op since `A.[[Status]]` is already linking. `B.[[Status]]` itself remains linking when control gets back to *A* and `InnerModuleLinking` is triggered on *C*. After this returns with `C.[[Status]]` being linked, both *A* and *B* transition from linking to linked together; this is by design, since they form a strongly connected component.

An analogous story occurs for the evaluation phase of a cyclic module graph, in the success case.

Now consider a case where *A* has an linking error; for example, it tries to import a binding from *C* that does not exist. In that case, the above steps still occur, including the early return from the second call to `InnerModuleLinking` on *A*. However, once we unwind back to the original `InnerModuleLinking` on *A*, it fails during `InitializeEnvironment`, namely right after `C.ResolveExport()`. The thrown `SyntaxError` exception propagates up to `A.Link`, which resets all modules that are currently on its *stack* (these are always exactly the modules that are still linking). Hence both *A* and *B* become unlinked. Note that *C* is left as linked.

Finally, consider a case where *A* has an evaluation error; for example, its source code throws an exception. In that case, the evaluation-time analog of the above steps still occurs, including the early return from the second call to `InnerModuleEvaluation` on *A*. However, once we unwind back to the original `InnerModuleEvaluation` on *A*, it fails by assumption. The exception thrown propagates up to `A.Evaluate()`, which records the error in all modules that are currently on its *stack* (i.e., the modules that are still evaluating). Hence both *A* and *B* become evaluated and the exception is recorded in both *A* and *B*'s `[[EvaluationError]]` fields, while *C* is left as evaluated with no `[[EvaluationError]]`.

16.2.1.6 Source Text Module Records

A *Source Text Module Record* is used to represent information about a module that was defined from ECMAScript source text (11) that was parsed using the `goal symbol Module`. Its fields contain digested information about the names that are imported by the module and its concrete methods use this digest to link, link, and evaluate the module.

A *Source Text Module Record* can exist in a module graph with other subclasses of the abstract `Module Record` type, and can participate in cycles with other subclasses of the `Cyclic Module Record` type.

In addition to the fields defined in Table 42, Source Text Module Records have the additional fields listed in Table 44. Each of these fields is initially set in `ParseModule`.

Table 44: Additional Fields of Source Text Module Records

Field Name	Value Type	Meaning
[[ECMAScriptCode]]	a Parse Node	The result of parsing the source text of this module using <i>Module</i> as the goal symbol.
[[Context]]	An ECMAScript execution context .	The execution context associated with this module.
[[ImportMeta]]	Object	An object exposed through the import.meta meta property. It is empty until it is accessed by ECMAScript code.
[[ImportEntries]]	List of ImportEntry Records	A List of ImportEntry records derived from the code of this module.
[[LocalExportEntries]]	List of ExportEntry Records	A List of ExportEntry records derived from the code of this module that correspond to declarations that occur within the module.
[[IndirectExportEntries]]	List of ExportEntry Records	A List of ExportEntry records derived from the code of this module that correspond to reexported imports that occur within the module or exports from export * as namespace declarations.
[[StarExportEntries]]	List of ExportEntry Records	A List of ExportEntry records derived from the code of this module that correspond to export * declarations that occur within the module, not including export * as namespace declarations.

An *ImportEntry Record* is a [Record](#) that digests information about a single declarative import. Each [ImportEntry Record](#) has the fields defined in [Table 45](#):

Table 45: [ImportEntry Record](#) Fields

Field Name	Value Type	Meaning
[[ModuleRequest]]	String	String value of the <i>ModuleSpecifier</i> of the <i>ImportDeclaration</i> .
[[ImportName]]	String	The name under which the desired binding is exported by the module identified by [[ModuleRequest]]. The value "" indicates that the import request is for the target module's namespace object.
[[LocalName]]	String	The name that is used to locally access the imported value from within the importing module.

Table 46 gives examples of `ImportEntry` records fields used to represent the syntactic import forms:

Table 46 (Informative): Import Forms Mappings to `ImportEntry` Records

Import Statement Form	[[ModuleRequest]]	[[ImportName]]	[[LocalName]]
<code>import v from "mod";</code>	"mod"	"default"	"v"
<code>import * as ns from "mod";</code>	"mod"	"*"	"ns"
<code>import {x} from "mod";</code>	"mod"	"x"	"x"
<code>import {x as v} from "mod";</code>	"mod"	"x"	"v"
<code>import "mod";</code>	An <code>ImportEntry Record</code> is not created.		

An `ExportEntry Record` is a `Record` that digests information about a single declarative export. Each `ExportEntry Record` has the fields defined in Table 47:

Table 47: `ExportEntry Record` Fields

Field Name	Value Type	Meaning
[[ExportName]]	String null	The name used to export this binding by this module.
[[ModuleRequest]]	String null	The String value of the <code>ModuleSpecifier</code> of the <code>ExportDeclaration</code> . null if the <code>ExportDeclaration</code> does not have a <code>ModuleSpecifier</code> .
[[ImportName]]	String null	The name under which the desired binding is exported by the module identified by [[ModuleRequest]]. null if the <code>ExportDeclaration</code> does not have a <code>ModuleSpecifier</code> . "*" indicates that the export request is for all exported bindings.
[[LocalName]]	String null	The name that is used to locally access the exported value from within the importing module. null if the exported value is not locally accessible from within the module.

Table 48 gives examples of the ExportEntry record fields used to represent the syntactic export forms:

Table 48 (Informative): Export Forms Mappings to ExportEntry Records

Export Statement Form	[[ExportName]]	[[ModuleRequest]]	[[ImportName]]	[[LocalName]]
export var v;	"v"	null	null	"v"
export default function f() {}	"default"	null	null	"f"
export default function () {}	"default"	null	null	"*default*"
export default 42;	"default"	null	null	"*default*"
export {x};	"x"	null	null	"x"
export {v as x};	"x"	null	null	"v"
export {x} from "mod";	"x"	"mod"	"x"	null
export {v as x} from "mod";	"x"	"mod"	"v"	null
export * from "mod";	null	"mod"	"*"	null
export * as ns from "mod";	"ns"	"mod"	"*"	null

The following definitions specify the required concrete methods and other [abstract operations](#) for Source Text Module Records

16.2.1.6.1 ParseModule (*sourceText*, *realm*, *hostDefined*)

The abstract operation ParseModule takes arguments *sourceText* (ECMAScript source text), *realm*, and *hostDefined*. It creates a [Source Text Module Record](#) based upon the result of parsing *sourceText* as a *Module*. It performs the following

steps when called:

1. **Assert**: *sourceText* is an ECMAScript source text (see clause 11).
2. Let *body* be `ParseText(sourceText, Module)`.
3. If *body* is a **List** of errors, return *body*.
4. Let *requestedModules* be the **ModuleRequests** of *body*.
5. Let *importEntries* be **ImportEntries** of *body*.
6. Let *importedBoundNames* be **ImportedLocalNames(importEntries)**.
7. Let *indirectExportEntries* be a new empty **List**.
8. Let *localExportEntries* be a new empty **List**.
9. Let *starExportEntries* be a new empty **List**.
10. Let *exportEntries* be **ExportEntries** of *body*.
11. For each **ExportEntry Record** *ee* of *exportEntries*, do
 - a. If *ee*.[[ModuleRequest]] is **null**, then
 - i. If *ee*.[[LocalName]] is not an element of *importedBoundNames*, then
 1. Append *ee* to *localExportEntries*.
 - ii. Else,
 1. Let *ie* be the element of *importEntries* whose [[LocalName]] is the same as *ee*.[[LocalName]].
 2. If *ie*.[[ImportName]] is **"*"**, then
 - a. NOTE: This is a re-export of an imported module namespace object.
 - b. Append *ee* to *localExportEntries*.
 3. Else,
 - a. NOTE: This is a re-export of a single name.
 - b. Append the **ExportEntry Record** { [[ModuleRequest]]: *ie*.[[ModuleRequest]], [[ImportName]]: *ie*.[[ImportName]], [[LocalName]]: **null**, [[ExportName]]: *ee*.[[ExportName]] } to *indirectExportEntries*.
 - b. Else if *ee*.[[ImportName]] is **"*"** and *ee*.[[ExportName]] is **null**, then
 - i. Append *ee* to *starExportEntries*.
 - c. Else,
 - i. Append *ee* to *indirectExportEntries*.
 12. Return **Source Text Module Record** { [[Realm]]: *realm*, [[Environment]]: **undefined**, [[Namespace]]: **undefined**, [[Status]]: **unlinked**, [[EvaluationError]]: **undefined**, [[HostDefined]]: *hostDefined*, [[ECMAScriptCode]]: *body*, [[Context]]: **empty**, [[ImportMeta]]: **empty**, [[RequestedModules]]: *requestedModules*, [[ImportEntries]]: *importEntries*, [[LocalExportEntries]]: *localExportEntries*, [[IndirectExportEntries]]: *indirectExportEntries*, [[StarExportEntries]]: *starExportEntries*, [[DFSIndex]]: **undefined**, [[DFSAncestorIndex]]: **undefined** }.

NOTE

An implementation may parse module source text and analyse it for Early Error conditions prior to the evaluation of `ParseModule` for that module source text. However, the reporting of any errors must be deferred until the point where this specification actually performs `ParseModule` upon that source text.

16.2.1.6.2 GetExportedNames ([*exportStarSet*]) Concrete Method

The `GetExportedNames` concrete method of a **Source Text Module Record** *module* takes optional argument *exportStarSet*. It performs the following steps when called:

1. If *exportStarSet* is not present, set *exportStarSet* to a new empty **List**.

2. **Assert:** *exportStarSet* is a **List** of Source Text Module Records.
3. If *exportStarSet* contains *module*, then
 - a. **Assert:** We've reached the starting point of an **export *** circularity.
 - b. Return a new empty **List**.
4. Append *module* to *exportStarSet*.
5. Let *exportedNames* be a new empty **List**.
6. For each **ExportEntry Record** *e* of *module*.**[[LocalExportEntries]]**, do
 - a. **Assert:** *module* provides the direct binding for this export.
 - b. Append *e*.**[[ExportName]]** to *exportedNames*.
7. For each **ExportEntry Record** *e* of *module*.**[[IndirectExportEntries]]**, do
 - a. **Assert:** *module* imports a specific binding for this export.
 - b. Append *e*.**[[ExportName]]** to *exportedNames*.
8. For each **ExportEntry Record** *e* of *module*.**[[StarExportEntries]]**, do
 - a. Let *requestedModule* be ?**HostResolveImportedModule**(*module*, *e*.**[[ModuleRequest]]**).
 - b. Let *starNames* be ?*requestedModule*.**GetExportedNames**(*exportStarSet*).
 - c. For each element *n* of *starNames*, do
 - i. If **SameValue**(*n*, "default") is **false**, then
 1. If *n* is not an element of *exportedNames*, then
 - a. Append *n* to *exportedNames*.
9. Return *exportedNames*.

NOTE **GetExportedNames** does not filter out or throw an exception for names that have ambiguous star export bindings.

16.2.1.6.3 **ResolveExport** (*exportName* [, *resolveSet*]) **Concrete Method**

The **ResolveExport** concrete method of a **Source Text Module Record** *module* takes argument *exportName* (a String) and optional argument *resolveSet*.

ResolveExport attempts to resolve an imported binding to the actual defining module and local binding name. The defining module may be the module represented by the **Module Record** this method was invoked on or some other module that is imported by that module. The parameter *resolveSet* is used to detect unresolved circular import/export paths. If a pair consisting of specific **Module Record** and *exportName* is reached that is already in *resolveSet*, an import circularity has been encountered. Before recursively calling **ResolveExport**, a pair consisting of *module* and *exportName* is added to *resolveSet*.

If a defining module is found, a **ResolvedBinding Record** { **[[Module]]**, **[[BindingName]]** } is returned. This record identifies the resolved binding of the originally requested export, unless this is the export of a namespace with no local binding. In this case, **[[BindingName]]** will be set to **"*namespace*"**. If no definition was found or the request is found to be circular, **null** is returned. If the request is found to be ambiguous, the string **"ambiguous"** is returned.

This concrete method performs the following steps when called:

1. If *resolveSet* is not present, set *resolveSet* to a new empty **List**.
2. **Assert:** *resolveSet* is a **List** of **Record** { **[[Module]]**, **[[ExportName]]** }.
3. For each **Record** { **[[Module]]**, **[[ExportName]]** } *r* of *resolveSet*, do
 - a. If *module* and *r*.**[[Module]]** are the same **Module Record** and **SameValue**(*exportName*, *r*.**[[ExportName]]**) is **true**, then
 - i. **Assert:** This is a circular import request.

- ii. Return **null**.
- 4. Append the **Record** { [[Module]]: *module*, [[ExportName]]: *exportName* } to *resolveSet*.
- 5. For each **ExportEntry Record** *e* of *module*.[[LocalExportEntries]], do
 - a. If **SameValue**(*exportName*, *e*.[[ExportName]]) is **true**, then
 - i. **Assert**: *module* provides the direct binding for this export.
 - ii. Return **ResolvedBinding Record** { [[Module]]: *module*, [[BindingName]]: *e*.[[LocalName]] }.
- 6. For each **ExportEntry Record** *e* of *module*.[[IndirectExportEntries]], do
 - a. If **SameValue**(*exportName*, *e*.[[ExportName]]) is **true**, then
 - i. Let *importedModule* be ? **HostResolveImportedModule**(*module*, *e*.[[ModuleRequest]]).
 - ii. If *e*.[[ImportName]] is "", then
 - 1. **Assert**: *module* does not provide the direct binding for this export.
 - 2. Return **ResolvedBinding Record** { [[Module]]: *importedModule*, [[BindingName]]: **"*namespace*"** }.
 - iii. Else,
 - 1. **Assert**: *module* imports a specific binding for this export.
 - 2. Return *importedModule*.ResolveExport(*e*.[[ImportName]], *resolveSet*).
- 7. If **SameValue**(*exportName*, "default") is **true**, then
 - a. **Assert**: A **default** export was not explicitly defined by this module.
 - b. Return **null**.
 - c. NOTE: A **default** export cannot be provided by an **export *** or **export * from "mod"** declaration.
- 8. Let *starResolution* be **null**.
- 9. For each **ExportEntry Record** *e* of *module*.[[StarExportEntries]], do
 - a. Let *importedModule* be ? **HostResolveImportedModule**(*module*, *e*.[[ModuleRequest]]).
 - b. Let *resolution* be ? *importedModule*.ResolveExport(*exportName*, *resolveSet*).
 - c. If *resolution* is "ambiguous", return "ambiguous".
 - d. If *resolution* is not **null**, then
 - i. **Assert**: *resolution* is a **ResolvedBinding Record**.
 - ii. If *starResolution* is **null**, set *starResolution* to *resolution*.
 - iii. Else,
 - 1. **Assert**: There is more than one * import that includes the requested name.
 - 2. If *resolution*.[[Module]] and *starResolution*.[[Module]] are not the same **Module Record** or **SameValue**(*resolution*.[[BindingName]], *starResolution*.[[BindingName]]) is **false**, return "ambiguous".
- 10. Return *starResolution*.

16.2.1.6.4 InitializeEnvironment () Concrete Method

The InitializeEnvironment concrete method of a **Source Text Module Record** *module* takes no arguments. It performs the following steps when called:

- 1. For each **ExportEntry Record** *e* of *module*.[[IndirectExportEntries]], do
 - a. Let *resolution* be ? *module*.ResolveExport(*e*.[[ExportName]]).
 - b. If *resolution* is **null** or "ambiguous", throw a **SyntaxError** exception.
 - c. **Assert**: *resolution* is a **ResolvedBinding Record**.
- 2. **Assert**: All named exports from *module* are resolvable.
- 3. Let *realm* be *module*.[[Realm]].
- 4. **Assert**: *realm* is not **undefined**.

5. Let *env* be `NewModuleEnvironment(realm.[[GlobalEnv]])`.
6. Set *module*.`[[Environment]]` to *env*.
7. For each `ImportEntry Record` *in* of *module*.`[[ImportEntries]]`, do
 - a. Let *importedModule* be ! `HostResolveImportedModule(module, in.[[ModuleRequest]])`.
 - b. NOTE: The above call cannot fail because imported module requests are a subset of *module*.`[[RequestedModules]]`, and these have been resolved earlier in this algorithm.
 - c. If *in*.`[[ImportName]]` is `"*"`, then
 - i. Let *namespace* be ? `GetModuleNamespace(importedModule)`.
 - ii. Perform ! *env*.`CreateImmutableBinding(in.[[LocalName]], true)`.
 - iii. Call *env*.`InitializeBinding(in.[[LocalName]], namespace)`.
 - d. Else,
 - i. Let *resolution* be ? *importedModule*.`ResolveExport(in.[[ImportName]])`.
 - ii. If *resolution* is `null` or `"ambiguous"`, throw a `SyntaxError` exception.
 - iii. If *resolution*.`[[BindingName]]` is `"* namespace*"`, then
 1. Let *namespace* be ? `GetModuleNamespace(resolution.[[Module]])`.
 2. Perform ! *env*.`CreateImmutableBinding(in.[[LocalName]], true)`.
 3. Call *env*.`InitializeBinding(in.[[LocalName]], namespace)`.
 - iv. Else,
 1. Call *env*.`CreateImportBinding(in.[[LocalName]], resolution.[[Module]], resolution.[[BindingName]])`.
8. Let *moduleContext* be a new ECMAScript code `execution context`.
9. Set the Function of *moduleContext* to `null`.
10. **Assert:** *module*.`[[Realm]]` is not `undefined`.
11. Set the `Realm` of *moduleContext* to *module*.`[[Realm]]`.
12. Set the `ScriptOrModule` of *moduleContext* to *module*.
13. Set the `VariableEnvironment` of *moduleContext* to *module*.`[[Environment]]`.
14. Set the `LexicalEnvironment` of *moduleContext* to *module*.`[[Environment]]`.
15. Set *module*.`[[Context]]` to *moduleContext*.
16. Push *moduleContext* onto the `execution context stack`; *moduleContext* is now the `running execution context`.
17. Let *code* be *module*.`[[ECMAScriptCode]]`.
18. Let *varDeclarations* be the `VarScopedDeclarations` of *code*.
19. Let *declaredVarNames* be a new empty `List`.
20. For each element *d* of *varDeclarations*, do
 - a. For each element *dn* of the `BoundNames` of *d*, do
 - i. If *dn* is not an element of *declaredVarNames*, then
 1. Perform ! *env*.`CreateMutableBinding(dn, false)`.
 2. Call *env*.`InitializeBinding(dn, undefined)`.
 3. Append *dn* to *declaredVarNames*.
21. Let *lexDeclarations* be the `LexicallyScopedDeclarations` of *code*.
22. For each element *d* of *lexDeclarations*, do
 - a. For each element *dn* of the `BoundNames` of *d*, do
 - i. If `IsConstantDeclaration` of *d* is `true`, then
 1. Perform ! *env*.`CreateImmutableBinding(dn, true)`.
 - ii. Else,
 1. Perform ! *env*.`CreateMutableBinding(dn, false)`.
 - iii. If *d* is a `FunctionDeclaration`, a `GeneratorDeclaration`, an `AsyncFunctionDeclaration`, or an `AsyncGeneratorDeclaration`, then
 1. Let *fo* be `InstantiateFunctionObject` of *d* with argument *env*.

2. Call `env.InitializeBinding(dn, fo)`.
23. Remove `moduleContext` from the `execution context stack`.
24. Return `NormalCompletion(empty)`.

16.2.1.6.5 ExecuteModule () Concrete Method

The `ExecuteModule` concrete method of a `Source Text Module Record` *module* takes no arguments. It performs the following steps when called:

1. Suspend the currently `running execution context`.
2. Let `moduleContext` be `module.[[Context]]`.
3. Push `moduleContext` onto the `execution context stack`; `moduleContext` is now the `running execution context`.
4. Let `result` be the result of evaluating `module.[[ECMAScriptCode]]`.
5. Suspend `moduleContext` and remove it from the `execution context stack`.
6. Resume the context that is now on the top of the `execution context stack` as the `running execution context`.
7. Return `Completion(result)`.

16.2.1.7 HostResolveImportedModule (*referencingScriptOrModule*, *specifier*)

The `host-defined` abstract operation `HostResolveImportedModule` takes arguments *referencingScriptOrModule* (a `Script Record` or `Module Record` or `null`) and *specifier* (a `ModuleSpecifier String`). It provides the concrete `Module Record` subclass instance that corresponds to *specifier* occurring within the context of the script or module represented by *referencingScriptOrModule*. *referencingScriptOrModule* may be `null` if the resolution is being performed in the context of an `import()` expression and there is no `active script or module` at that time.

NOTE

An example of when *referencingScriptOrModule* can be `null` is in a web browser `host`. There, if a user clicks on a control given by

```
<button type="button" onclick="import('./foo.mjs')">Click me</button>
```

there will be no `active script or module` at the time the `import()` expression runs. More generally, this can happen in any situation where the `host` pushes execution contexts with `null` `ScriptOrModule` components onto the `execution context stack`.

The implementation of `HostResolveImportedModule` must conform to the following requirements:

- The normal return value must be an instance of a concrete subclass of `Module Record`.
- If a `Module Record` corresponding to the pair *referencingScriptOrModule*, *specifier* does not exist or cannot be created, an exception must be thrown.
- Each time this operation is called with a specific *referencingScriptOrModule*, *specifier* pair as arguments it must return the same `Module Record` instance if it completes normally.

Multiple different *referencingScriptOrModule*, *specifier* pairs may map to the same `Module Record` instance. The actual mapping semantic is `host-defined` but typically a normalization process is applied to *specifier* as part of the mapping process. A typical normalization process would include actions such as alphabetic case folding and expansion of relative and abbreviated path specifiers.

16.2.1.8 HostImportModuleDynamically (*referencingScriptOrModule*, *specifier*, *promiseCapability*)

The **host-defined** abstract operation `HostImportModuleDynamically` takes arguments *referencingScriptOrModule* (a `Script Record` or `Module Record` or **null**), *specifier* (a `ModuleSpecifier String`), and *promiseCapability* (a `PromiseCapability Record`). It performs any necessary setup work in order to make available the module corresponding to *specifier* occurring within the context of the script or module represented by *referencingScriptOrModule*. *referencingScriptOrModule* may be **null** if there is no **active script or module** when the **import()** expression occurs. It then performs `FinishDynamicImport` to finish the dynamic import process.

The implementation of `HostImportModuleDynamically` must conform to the following requirements:

- The abstract operation must always complete normally with **undefined**. Success or failure must instead be signaled as discussed below.
- The **host environment** must conform to one of the two following sets of requirements:

Success path

- At some future time, the **host environment** must perform `FinishDynamicImport(referencingScriptOrModule, specifier, promiseCapability, NormalCompletion(undefined))`.
- Any subsequent call to `HostResolveImportedModule` after `FinishDynamicImport` has completed, given the arguments *referencingScriptOrModule* and *specifier*, must complete normally.
- The completion value of any subsequent call to `HostResolveImportedModule` after `FinishDynamicImport` has completed, given the arguments *referencingScriptOrModule* and *specifier*, must be a module which has already been evaluated, i.e. whose `Evaluate concrete` method has already been called and returned a normal completion.

Failure path

- At some future time, the **host environment** must perform `FinishDynamicImport(referencingScriptOrModule, specifier, promiseCapability, an abrupt completion)`, with the **abrupt completion** representing the cause of failure.
- If the **host environment** takes the success path once for a given *referencingScriptOrModule, specifier* pair, it must always do so for subsequent calls.
- The operation must not call *promiseCapability*.`[[Resolve]]` or *promiseCapability*.`[[Reject]]`, but instead must treat *promiseCapability* as an opaque identifying value to be passed through to `FinishDynamicImport`.

The actual process performed is **host-defined**, but typically consists of performing whatever I/O operations are necessary to allow `HostResolveImportedModule` to synchronously retrieve the appropriate `Module Record`, and then calling its `Evaluate concrete` method. This might require performing similar normalization as `HostResolveImportedModule` does.

16.2.1.9 `FinishDynamicImport` (*referencingScriptOrModule, specifier, promiseCapability, completion*)

The abstract operation `FinishDynamicImport` takes arguments *referencingScriptOrModule, specifier, promiseCapability* (a `PromiseCapability Record`), and *completion*. `FinishDynamicImport` completes the process of a dynamic import originally started by an **import()** call, resolving or rejecting the promise returned by that call as appropriate according to *completion*. It is performed by **host** environments as part of `HostImportModuleDynamically`. It performs the following steps when called:

1. If *completion* is an **abrupt completion**, perform ! `Call(promiseCapability. [[Reject]], undefined, « completion. [[Value]] »)`.
2. Else,
 - a. **Assert**: *completion* is a normal completion and *completion*.`[[Value]]` is **undefined**.

- b. Let *moduleRecord* be ! [HostResolveImportedModule](#)(*referencingScriptOrModule*, *specifier*).
- c. **Assert**: Evaluate has already been invoked on *moduleRecord* and successfully completed.
- d. Let *namespace* be [GetModuleNamespace](#)(*moduleRecord*).
- e. If *namespace* is an **abrupt completion**, perform ! [Call](#)(*promiseCapability*.[[Reject]], **undefined**, « *namespace*.[[Value]] »).
- f. Else, perform ! [Call](#)(*promiseCapability*.[[Resolve]], **undefined**, « *namespace*.[[Value]] »).

16.2.1.10 GetModuleNamespace (*module*)

The abstract operation [GetModuleNamespace](#) takes argument *module*. It retrieves the Module Namespace Object representing *module*'s exports, lazily creating it the first time it was requested, and storing it in *module*.[[Namespace]] for future retrieval. It performs the following steps when called:

1. **Assert**: *module* is an instance of a concrete subclass of [Module Record](#).
2. **Assert**: If *module* is a [Cyclic Module Record](#), then *module*.[[Status]] is not **unlinked**.
3. Let *namespace* be *module*.[[Namespace]].
4. If *namespace* is **undefined**, then
 - a. Let *exportedNames* be ? *module*.GetExportedNames().
 - b. Let *unambiguousNames* be a new empty [List](#).
 - c. For each element *name* of *exportedNames*, do
 - i. Let *resolution* be ? *module*.ResolveExport(*name*).
 - ii. If *resolution* is a [Resolved Binding Record](#), append *name* to *unambiguousNames*.
 - d. Set *namespace* to [ModuleNamespaceCreate](#)(*module*, *unambiguousNames*).
5. Return *namespace*.

NOTE The only way [GetModuleNamespace](#) can throw is via one of the triggered [HostResolveImportedModule](#) calls. Unresolvable names are simply excluded from the namespace at this point. They will lead to a real linking error later unless they are all ambiguous star exports that are not explicitly requested anywhere.

16.2.1.11 Runtime Semantics: Evaluation

Module : [empty]

1. Return [NormalCompletion](#)(**undefined**).

ModuleBody : *ModuleItemList*

1. Let *result* be the result of evaluating *ModuleItemList*.
2. If *result*.[[Type]] is normal and *result*.[[Value]] is empty, then
 - a. Return [NormalCompletion](#)(**undefined**).
3. Return [Completion](#)(*result*).

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *sl* be the result of evaluating *ModuleItemList*.
2. [ReturnIfAbrupt](#)(*sl*).
3. Let *s* be the result of evaluating *ModuleItem*.
4. Return [Completion](#)([UpdateEmpty](#)(*s*, *sl*)).

ModuleItem : *ImportDeclaration*

1. Return `NormalCompletion(empty)`.

16.2.2 Imports

Syntax

ImportDeclaration :

```
import ImportClause FromClause ;
import ModuleSpecifier ;
```

ImportClause :

```
ImportedDefaultBinding
NamespaceImport
NamedImports
ImportedDefaultBinding , NamespaceImport
ImportedDefaultBinding , NamedImports
```

ImportedDefaultBinding :

```
ImportedBinding
```

NamespaceImport :

```
* as ImportedBinding
```

NamedImports :

```
{ }
{ ImportsList }
{ ImportsList , }
```

FromClause :

```
from ModuleSpecifier
```

ImportsList :

```
ImportSpecifier
ImportsList , ImportSpecifier
```

ImportSpecifier :

```
ImportedBinding
IdentifierName as ImportedBinding
```

ModuleSpecifier :

```
StringLiteral
```

ImportedBinding :

```
BindingIdentifier [~Yield, ~Await]
```

16.2.2.1 Static Semantics: Early Errors

ModuleItem : *ImportDeclaration*

- It is a Syntax Error if the [BoundNames](#) of *ImportDeclaration* contains any duplicate entries.

16.2.2.2 Static Semantics: ImportEntries

Module : [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](#).

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](#).

16.2.2.3 Static Semantics: ImportEntriesForModule

With parameter *module*.

ImportClause : *ImportedDefaultBinding* , *NameSpaceImport*

1. Let *entries* be [ImportEntriesForModule](#) of *ImportedDefaultBinding* with argument *module*.
2. Append to *entries* the elements of the [ImportEntriesForModule](#) of *NameSpaceImport* with argument *module*.
3. Return *entries*.

ImportClause : *ImportedDefaultBinding* , *NamedImports*

1. Let *entries* be [ImportEntriesForModule](#) of *ImportedDefaultBinding* with argument *module*.
2. Append to *entries* the elements of the [ImportEntriesForModule](#) of *NamedImports* with argument *module*.
3. Return *entries*.

ImportedDefaultBinding : *ImportedBinding*

1. Let *localName* be the sole element of [BoundNames](#) of *ImportedBinding*.
2. Let *defaultEntry* be the [ImportEntry Record](#) { [\[\[ModuleRequest\]\]](#): *module*, [\[\[ImportName\]\]](#): "default", [\[\[LocalName\]\]](#): *localName* }.
3. Return a [List](#) whose sole element is *defaultEntry*.

NamespaceImport : * **as** *ImportedBinding*

1. Let *localName* be the [StringValue](#) of *ImportedBinding*.
2. Let *entry* be the [ImportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: "*", [[LocalName]]: *localName* }.
3. Return a [List](#) whose sole element is *entry*.

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 sole element of [BoundNames](#) of *ImportedBinding*.
2. Let *entry* be the [ImportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: *localName*, [[LocalName]]: *localName* }.
3. Return a [List](#) whose sole element is *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 [ImportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: *importName*, [[LocalName]]: *localName* }.
4. Return a [List](#) whose sole element is *entry*.

16.2.3 Exports

Syntax

ExportDeclaration :

```
export ExportFromClause FromClause ;  
export NamedExports ;  
export VariableStatement [~Yield, ~Await]  
export Declaration [~Yield, ~Await]  
export default HoistableDeclaration [~Yield, ~Await, +Default]  
export default ClassDeclaration [~Yield, ~Await, +Default]  
export default [lookahead ∉ { function, async [no LineTerminator here] function, class }]  
    AssignmentExpression [+In, ~Yield, ~Await] ;
```

ExportFromClause :

```
*  
* as IdentifierName  
NamedExports
```

NamedExports :

```
{ }  
{ ExportsList }  
{ ExportsList , }
```

ExportsList :

```
ExportSpecifier  
ExportsList , ExportSpecifier
```

ExportSpecifier :

```
IdentifierName  
IdentifierName as IdentifierName
```

16.2.3.1 Static Semantics: Early Errors

ExportDeclaration : **export** *NamedExports* ;

- For each *IdentifierName* *n* in [ReferencedBindings](#) of *NamedExports*: It is a Syntax Error if [StringValue](#) of *n* is a [ReservedWord](#) or if the [StringValue](#) of *n* is one of: "implements", "interface", "let", "package", "private", "protected", "public", or "static".

NOTE The above rule means that each [ReferencedBindings](#) of *NamedExports* is treated as an [IdentifierReference](#).

16.2.3.2 Static Semantics: ExportedBindings

NOTE [ExportedBindings](#) are the locally bound names that are explicitly associated with a *Module*'s [ExportedNames](#).

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *names* be [ExportedBindings](#) of *ModuleItemList*.
2. Append to *names* the elements of the [ExportedBindings](#) of *ModuleItem*.
3. Return *names*.

ModuleItem :

```
ImportDeclaration  
StatementListItem
```

1. Return a new empty [List](#).

ExportDeclaration :

```
export ExportFromClause FromClause ;
```

1. Return a new empty [List](#).

ExportDeclaration : **export** *NamedExports* ;

1. Return the [ExportedBindings](#) of *NamedExports*.

ExportDeclaration : **export** *VariableStatement*

1. Return the **BoundNames** of *VariableStatement*.

ExportDeclaration : **export** *Declaration*

1. Return the **BoundNames** of *Declaration*.

ExportDeclaration :

export default *HoistableDeclaration*

export default *ClassDeclaration*

export default *AssignmentExpression* ;

1. Return the **BoundNames** of this *ExportDeclaration*.

NamedExports : { }

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 : *IdentifierName*

1. Return a **List** whose sole element is the **StringValue** of *IdentifierName*.

ExportSpecifier : *IdentifierName* **as** *IdentifierName*

1. Return a **List** whose sole element is the **StringValue** of the first *IdentifierName*.

16.2.3.3 Static Semantics: ExportedNames

NOTE ExportedNames are the externally visible names that a *Module* explicitly maps to one of its local name bindings.

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *names* be **ExportedNames** of *ModuleItemList*.
2. Append to *names* the elements of the **ExportedNames** of *ModuleItem*.
3. Return *names*.

ModuleItem : *ExportDeclaration*

1. Return the **ExportedNames** of *ExportDeclaration*.

ModuleItem :

ImportDeclaration

StatementListItem

1. Return a new empty [List](#).

ExportDeclaration : **export** *ExportFromClause* *FromClause* ;

1. Return the [ExportedNames](#) of *ExportFromClause*.

ExportFromClause : *

1. Return a new empty [List](#).

ExportFromClause : * **as** *IdentifierName*

1. Return a [List](#) whose sole element is the [StringValue](#) of *IdentifierName*.

ExportFromClause : *NamedExports*

1. Return the [ExportedNames](#) of *NamedExports*.

ExportDeclaration : **export** *VariableStatement*

1. Return the [BoundNames](#) of *VariableStatement*.

ExportDeclaration : **export** *Declaration*

1. Return the [BoundNames](#) of *Declaration*.

ExportDeclaration :

export default *HoistableDeclaration*

export default *ClassDeclaration*

export default *AssignmentExpression* ;

1. Return « "default" ».

NamedExports : { }

1. Return a new empty [List](#).

ExportsList : *ExportsList* , *ExportSpecifier*

1. Let *names* be the [ExportedNames](#) of *ExportsList*.
2. Append to *names* the elements of the [ExportedNames](#) of *ExportSpecifier*.
3. Return *names*.

ExportSpecifier : *IdentifierName*

1. Return a [List](#) whose sole element is the [StringValue](#) of *IdentifierName*.

ExportSpecifier : *IdentifierName* **as** *IdentifierName*

1. Return a [List](#) whose sole element is the [StringValue](#) of the second *IdentifierName*.

16.2.3.4 Static Semantics: ExportEntries

Module : [empty]

1. Return a new empty [List](#).

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *entries* be *ExportEntries* of *ModuleItemList*.
2. Append to *entries* the elements of the *ExportEntries* of *ModuleItem*.
3. Return *entries*.

ModuleItem :

ImportDeclaration
StatementListItem

1. Return a new empty *List*.

ExportDeclaration : **export** *ExportFromClause* *FromClause* ;

1. Let *module* be the sole element of *ModuleRequests* of *FromClause*.
2. Return *ExportEntriesForModule* of *ExportFromClause* with argument *module*.

ExportDeclaration : **export** *NamedExports* ;

1. Return *ExportEntriesForModule* of *NamedExports* with argument **null**.

ExportDeclaration : **export** *VariableStatement*

1. Let *entries* be a new empty *List*.
2. Let *names* be the *BoundNames* of *VariableStatement*.
3. For each element *name* of *names*, do
 - a. Append the *ExportEntry Record* { *[[ModuleRequest]]*: **null**, *[[ImportName]]*: **null**, *[[LocalName]]*: *name*, *[[ExportName]]*: *name* } to *entries*.
4. Return *entries*.

ExportDeclaration : **export** *Declaration*

1. Let *entries* be a new empty *List*.
2. Let *names* be the *BoundNames* of *Declaration*.
3. For each element *name* of *names*, do
 - a. Append the *ExportEntry Record* { *[[ModuleRequest]]*: **null**, *[[ImportName]]*: **null**, *[[LocalName]]*: *name*, *[[ExportName]]*: *name* } to *entries*.
4. Return *entries*.

ExportDeclaration : **export default** *HoistableDeclaration*

1. Let *names* be *BoundNames* of *HoistableDeclaration*.
2. Let *localName* be the sole element of *names*.
3. Return a *List* whose sole element is the *ExportEntry Record* { *[[ModuleRequest]]*: **null**, *[[ImportName]]*: **null**, *[[LocalName]]*: *localName*, *[[ExportName]]*: **"default"** }.

ExportDeclaration : **export default** *ClassDeclaration*

1. Let *names* be *BoundNames* of *ClassDeclaration*.
2. Let *localName* be the sole element of *names*.
3. Return a *List* whose sole element is the *ExportEntry Record* { *[[ModuleRequest]]*: **null**, *[[ImportName]]*: **null**, *[[LocalName]]*: *localName*, *[[ExportName]]*: **"default"** }.

ExportDeclaration : **export default** *AssignmentExpression* ;

1. Let *entry* be the [ExportEntry Record](#) { [[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: **"*default"**, [[ExportName]]: **"default"** }.
2. Return a [List](#) whose sole element is *entry*.

NOTE **"*default"** is used within this specification as a synthetic name for anonymous default export values.

16.2.3.5 Static Semantics: ExportEntriesForModule

With parameter *module*.

ExportFromClause : *

1. Let *entry* be the [ExportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: **"*"**, [[LocalName]]: **null**, [[ExportName]]: **null** }.
2. Return a [List](#) whose sole element is *entry*.

ExportFromClause : * **as** *IdentifierName*

1. Let *exportName* be the [StringValue](#) of *IdentifierName*.
2. Let *entry* be the [ExportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: **"*"**, [[LocalName]]: **null**, [[ExportName]]: *exportName* }.
3. Return a [List](#) whose sole element is *entry*.

NamedExports : { }

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 : *IdentifierName*

1. Let *sourceName* be the [StringValue](#) of *IdentifierName*.
2. If *module* is **null**, then
 - a. Let *localName* be *sourceName*.
 - b. Let *importName* be **null**.
3. Else,
 - a. Let *localName* be **null**.
 - b. Let *importName* be *sourceName*.
4. Return a [List](#) whose sole element is the [ExportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: *importName*, [[LocalName]]: *localName*, [[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. If *module* is **null**, then
 - a. Let *localName* be *sourceName*.
 - b. Let *importName* be **null**.
4. Else,
 - a. Let *localName* be **null**.
 - b. Let *importName* be *sourceName*.
5. Return a [List](#) whose sole element is the [ExportEntry Record](#) { [[ModuleRequest]]: *module*, [[ImportName]]: *importName*, [[LocalName]]: *localName*, [[ExportName]]: *exportName* }.

16.2.3.6 Static Semantics: ReferencedBindings

NamedExports : { }

1. Return a new empty [List](#).

ExportsList : *ExportsList* , *ExportSpecifier*

1. Let *names* be the [ReferencedBindings](#) of *ExportsList*.
2. Append to *names* the elements of the [ReferencedBindings](#) of *ExportSpecifier*.
3. Return *names*.

ExportSpecifier : *IdentifierName*

1. Return a [List](#) whose sole element is the *IdentifierName*.

ExportSpecifier : *IdentifierName* **as** *IdentifierName*

1. Return a [List](#) whose sole element is the first *IdentifierName*.

16.2.3.7 Runtime Semantics: Evaluation

ExportDeclaration :

export *ExportFromClause* *FromClause* ;
export *NamedExports* ;

1. Return [NormalCompletion](#)(empty).

ExportDeclaration : **export** *VariableStatement*

1. Return the result of evaluating *VariableStatement*.

ExportDeclaration : **export** *Declaration*

1. Return the result of evaluating *Declaration*.

ExportDeclaration : **export default** *HoistableDeclaration*

1. Return the result of evaluating *HoistableDeclaration*.

ExportDeclaration : **export default** *ClassDeclaration*

1. Let *value* be ? [BindingClassDeclarationEvaluation](#) of *ClassDeclaration*.
2. Let *className* be the sole element of [BoundNames](#) of *ClassDeclaration*.

3. If *className* is **"*default*"**, then
 - a. Let *env* be the [running execution context](#)'s `LexicalEnvironment`.
 - b. Perform ? `InitializeBoundName`(**"*default*"**, *value*, *env*).
4. Return `NormalCompletion`(empty).

ExportDeclaration : **export default** *AssignmentExpression* ;

1. If `IsAnonymousFunctionDefinition`(*AssignmentExpression*) is **true**, then
 - a. Let *value* be ? `NamedEvaluation` of *AssignmentExpression* with argument **"default"**.
2. Else,
 - a. Let *rhs* be the result of evaluating *AssignmentExpression*.
 - b. Let *value* be ? `GetValue`(*rhs*).
3. Let *env* be the [running execution context](#)'s `LexicalEnvironment`.
4. Perform ? `InitializeBoundName`(**"*default*"**, *value*, *env*).
5. Return `NormalCompletion`(empty).

17 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 parsing that *Script* in `ParseScript`. 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:

- Except as restricted in 17.1, a [host](#) or 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 [host-defined](#) behaviour instead of throwing **SyntaxError** when they encounter a [host-defined](#) extension to the script syntax or regular expression pattern or flag syntax.
- Except as restricted in 17.1, a [host](#) or 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 [host-defined](#) behaviour instead of throwing an error (such as **ReferenceError**).

17.1 Forbidden Extensions

An implementation must not extend this specification in the following ways:

- ECMAScript function objects defined using syntactic constructors in [strict mode code](#) must not be created with own properties named **"caller"** or **"arguments"**. Such own properties also must not be created for function objects defined using an *ArrowFunction*, *MethodDefinition*, *GeneratorDeclaration*, *GeneratorExpression*, *AsyncGeneratorDeclaration*, *AsyncGeneratorExpression*, *ClassDeclaration*, *ClassExpression*, *AsyncFunctionDeclaration*, *AsyncFunctionExpression*, or *AsyncArrowFunction* regardless of whether the definition is contained in [strict mode code](#). Built-in functions, strict functions created using the Function [constructor](#), generator functions created using the Generator [constructor](#), async functions created using the AsyncFunction [constructor](#), and functions created using the **bind** method also must not be created with such own properties.
- If an implementation extends any [function object](#) with an own property named **"caller"** the value of that property, as observed using `[[Get]]` or `[[GetOwnProperty]]`, must not be a [strict function](#) object. If it is an [accessor property](#), the function that is the value of the property's `[[Get]]` attribute must never return a [strict function](#) when called.
- Neither mapped nor unmapped arguments objects may be created with an own property named **"caller"**.
- The behaviour of built-in methods which are specified in ECMA-402, such as those named **toLocaleString**, must not be extended except as specified in ECMA-402.
- The RegExp pattern grammars in [22.2.1](#) and [B.1.4](#) must not be extended to recognize any of the source characters A-Z or a-z as *IdentityEscape*_[+U] when the `[U]` grammar parameter is present.
- The Syntactic Grammar must not be extended in any manner that allows the token `:` to immediately follow source text that matches the *BindingIdentifier* nonterminal symbol.
- When processing [strict mode code](#), the syntax of *NumericLiteral* must not be extended to include *LegacyOctalIntegerLiteral* and the syntax of *DecimalIntegerLiteral* must not be extended to include *NonOctalDecimalIntegerLiteral* as described in [B.1.1](#).
- *TemplateCharacter* must not be extended to include *LegacyOctalEscapeSequence* or *NonOctalDecimalEscapeSequence* as defined in [B.1.2](#).
- When processing [strict mode code](#), the extensions defined in [B.3.2](#), [B.3.3](#), [B.3.4](#), and [B.3.6](#) must not be supported.
- When parsing for the *Module goal symbol*, the lexical grammar extensions defined in [B.1.3](#) must not be supported.
- *ImportCall* must not be extended.

18 ECMAScript Standard Built-in Objects

There are certain built-in objects available whenever an ECMAScript *Script* or *Module* begins execution. One, the [global object](#), is part of the global 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 [10.3](#). 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 [Realm Record](#) of the [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 the properties of that [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. In the description of a particular function, the terms “**this** value” and “NewTarget” have the meanings given in [10.3](#).

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 1 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** ([20.2.3](#)), 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** ([20.1.3](#)), 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.

Each built-in function defined in this specification is created by calling the [CreateBuiltinFunction](#) abstract operation ([10.3.3](#)). The values of the *length* and *name* parameters are the initial values of the “**length**” and “**name**” properties as discussed below. The values of the *prefix* parameter are similarly discussed below.

Every built-in [function object](#), including constructors, has a “**length**” property whose value is a non-negative [integral Number](#). Unless otherwise specified, this value is equal to the number of required parameters shown in the subclause headings for the function description. Optional parameters and rest parameters are not included in the parameter count.

NOTE 2 For example, the [function object](#) that is the initial value of the “**map**” property of the [Array prototype object](#) is described under the subclause heading «`Array.prototype.map (callbackFn [, thisArg])`» which shows the two named arguments `callbackFn` and `thisArg`, the latter being optional; therefore the value of the “**length**” property of that [function object](#) is `1`.

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, 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. Functions that are identified as anonymous functions use the empty String as the value of the “**name**” property. 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**” (respectively) passed to the *prefix* parameter when calling [CreateBuiltinFunction](#).

The value of the **"name"** property is explicitly specified for each built-in functions whose property key is a Symbol value. If such an explicitly specified value starts with the prefix **"get "** or **"set "** and the function for which it is specified is a get or set accessor function of a built-in property, the value without the prefix is passed to the *name* parameter, and the value **"get"** or **"set"** (respectively) is passed to the *prefix* parameter when calling [CreateBuiltinFunction](#).

Unless otherwise specified, the **"name"** property of a built-in [function object](#) has the attributes { **[[Writable]]: false**, **[[Enumerable]]: false**, **[[Configurable]]: true** }.

Every other [data property](#) described in clauses 19 through 28 and in Annex B.2 has the attributes { **[[Writable]]: true**, **[[Enumerable]]: false**, **[[Configurable]]: true** } unless otherwise specified.

Every [accessor property](#) described in clauses 19 through 28 and in Annex B.2 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 described the get accessor is the default value, **undefined**.

19 The Global Object

The *global object*:

- is created before control enters any [execution context](#).
- does not have a **[[Construct]]** internal method; it cannot be used as a [constructor](#) with the **new** operator.
- does not have a **[[Call]]** internal method; it cannot be invoked as a function.
- has a **[[Prototype]]** internal slot whose value is [host-defined](#).
- may have [host](#) defined properties in addition to the properties defined in this specification. This may include a property whose value is the global object itself.

19.1 Value Properties of the Global Object

19.1.1 globalThis

The initial value of the **"globalThis"** property of the [global object](#) in a [Realm Record](#) *realm* is *realm*.**[[GlobalEnv]]**.**[[GlobalThisValue]]**.

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

19.1.2 Infinity

The value of **Infinity** is $+\infty_{\mathbb{F}}$ (see 6.1.6.1). This property has the attributes { **[[Writable]]: false**, **[[Enumerable]]: false**, **[[Configurable]]: false** }.

19.1.3 NaN

The value of **NaN** is NaN (see 6.1.6.1). This property has the attributes { **[[Writable]]: false**, **[[Enumerable]]: false**, **[[Configurable]]: false** }.

19.1.4 undefined

The value of **undefined** is **undefined** (see 6.1.1). This property has the attributes { **[[Writable]]**: **false**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **false** }.

19.2 Function Properties of the Global Object

19.2.1 eval (*x*)

The **eval** function is the *%eval%* intrinsic object. When the **eval** function is called with one argument *x*, the following steps are taken:

1. **Assert**: The **execution context stack** has at least two elements.
2. Let *callerContext* be the second to top element of the **execution context stack**.
3. Let *callerRealm* be *callerContext*'s **Realm**.
4. Return ? **PerformEval**(*x*, *callerRealm*, **false**, **false**).

19.2.1.1 PerformEval (*x*, *callerRealm*, *strictCaller*, *direct*)

The abstract operation **PerformEval** takes arguments *x*, *callerRealm*, *strictCaller*, and *direct*. It performs the following steps when called:

1. **Assert**: If *direct* is **false**, then *strictCaller* is also **false**.
2. If **Type**(*x*) is not **String**, return *x*.
3. Let *evalRealm* be the **current Realm Record**.
4. Perform ? **HostEnsureCanCompileStrings**(*callerRealm*, *evalRealm*).
5. Let *inFunction* be **false**.
6. Let *inMethod* be **false**.
7. Let *inDerivedConstructor* be **false**.
8. If *direct* is **true**, then
 - a. Let *thisEnvRec* be ! **GetThisEnvironment**(*x*).
 - b. If *thisEnvRec* is a **function Environment Record**, then
 - i. Let *F* be *thisEnvRec*.**[[FunctionObject]]**.
 - ii. Set *inFunction* to **true**.
 - iii. Set *inMethod* to *thisEnvRec*.**HasSuperBinding**(*x*).
 - iv. If *F*.**[[ConstructorKind]]** is **derived**, set *inDerivedConstructor* to **true**.
9. Perform the following substeps in an **implementation-defined** order, possibly interleaving parsing and error detection:
 - a. Let *script* be **ParseText**(! **StringToCodePoints**(*x*), *Script*).
 - b. If *script* is a **List** of errors, throw a **SyntaxError** exception.
 - c. If *script* **Contains** *ScriptBody* is **false**, return **undefined**.
 - d. Let *body* be the *ScriptBody* of *script*.
 - e. If *inFunction* is **false**, and *body* **Contains** *NewTarget*, throw a **SyntaxError** exception.
 - f. If *inMethod* is **false**, and *body* **Contains** *SuperProperty*, throw a **SyntaxError** exception.
 - g. If *inDerivedConstructor* is **false**, and *body* **Contains** *SuperCall*, throw a **SyntaxError** exception.
10. If *strictCaller* is **true**, let *strictEval* be **true**.
11. Else, let *strictEval* be **IsStrict** of *script*.

12. Let *runningContext* be the **running execution context**.
13. NOTE: If *direct* is **true**, *runningContext* will be the **execution context** that performed the **direct eval**. If *direct* is **false**, *runningContext* will be the **execution context** for the invocation of the **eval** function.
14. If *direct* is **true**, then
 - a. Let *lexEnv* be **NewDeclarativeEnvironment**(*runningContext*'s **LexicalEnvironment**).
 - b. Let *varEnv* be *runningContext*'s **VariableEnvironment**.
15. Else,
 - a. Let *lexEnv* be **NewDeclarativeEnvironment**(*evalRealm*.[[**GlobalEnv**]]).
 - b. Let *varEnv* be *evalRealm*.[[**GlobalEnv**]].
16. If *strictEval* is **true**, set *varEnv* to *lexEnv*.
17. If *runningContext* is not already suspended, suspend *runningContext*.
18. Let *evalContext* be a new ECMAScript code **execution context**.
19. Set *evalContext*'s **Function** to **null**.
20. Set *evalContext*'s **Realm** to *evalRealm*.
21. Set *evalContext*'s **ScriptOrModule** to *runningContext*'s **ScriptOrModule**.
22. Set *evalContext*'s **VariableEnvironment** to *varEnv*.
23. Set *evalContext*'s **LexicalEnvironment** to *lexEnv*.
24. Push *evalContext* onto the **execution context stack**; *evalContext* is now the **running execution context**.
25. Let *result* be **EvalDeclarationInstantiation**(*body*, *varEnv*, *lexEnv*, *strictEval*).
26. If *result*.[[**Type**]] is normal, then
 - a. Set *result* to the result of evaluating *body*.
27. If *result*.[[**Type**]] is normal and *result*.[[**Value**]] is empty, then
 - a. Set *result* to **NormalCompletion**(**undefined**).
28. Suspend *evalContext* and remove it from the **execution context stack**.
29. Resume the context that is now on the top of the **execution context stack** as the **running execution context**.
30. Return **Completion**(*result*).

NOTE The eval code cannot instantiate variable or function bindings in the variable environment of the calling context that invoked the eval if the calling context is evaluating formal parameter initializers or if either the code of the calling context or the eval code is **strict mode code**. Instead such bindings are instantiated in a new **VariableEnvironment** that is only accessible to the eval code. Bindings introduced by **let**, **const**, or **class** declarations are always instantiated in a new **LexicalEnvironment**.

19.2.1.2 HostEnsureCanCompileStrings (*callerRealm*, *calleeRealm*)

The **host-defined** abstract operation **HostEnsureCanCompileStrings** takes arguments *callerRealm* (a **Realm Record**) and *calleeRealm* (a **Realm Record**). It allows **host** environments to block certain ECMAScript functions which allow developers to compile strings into ECMAScript code.

An implementation of **HostEnsureCanCompileStrings** may complete normally or abruptly. Any abrupt completions will be propagated to its callers. The default implementation of **HostEnsureCanCompileStrings** is to unconditionally return an empty normal completion.

19.2.1.3 EvalDeclarationInstantiation (*body*, *varEnv*, *lexEnv*, *strict*)

The abstract operation **EvalDeclarationInstantiation** takes arguments *body*, *varEnv*, *lexEnv*, and *strict*. It performs the

following steps when called:

1. Let *varNames* be the [VarDeclaredNames](#) of *body*.
2. Let *varDeclarations* be the [VarScopedDeclarations](#) of *body*.
3. If *strict* is **false**, then
 - a. If *varEnv* is a [global Environment Record](#), then
 - i. For each element *name* of *varNames*, do
 1. If *varEnv*.[HasLexicalDeclaration](#)(*name*) is **true**, throw a **SyntaxError** exception.
 2. NOTE: **eval** will not create a global var declaration that would be shadowed by a global lexical declaration.
 - b. Let *thisEnv* be *lexEnv*.
 - c. **Assert**: The following loop will terminate.
 - d. Repeat, while *thisEnv* is not the same as *varEnv*,
 - i. If *thisEnv* is not an [object Environment Record](#), then
 1. NOTE: The environment of with statements cannot contain any lexical declaration so it doesn't need to be checked for var/let hoisting conflicts.
 2. For each element *name* of *varNames*, do
 - a. If *thisEnv*.[HasBinding](#)(*name*) is **true**, then
 - i. Throw a **SyntaxError** exception.
 - ii. NOTE: Annex [B.3.5](#) defines alternate semantics for the above step.
 - b. NOTE: A **direct eval** will not hoist var declaration over a like-named lexical declaration.
 - ii. Set *thisEnv* to *thisEnv*.[\[\[OuterEnv\]\]](#).
4. Let *functionsToInitialize* be a new empty [List](#).
5. Let *declaredFunctionNames* be a new empty [List](#).
6. For each element *d* of *varDeclarations*, in reverse [List](#) order, do
 - a. If *d* is neither a [VariableDeclaration](#) nor a [ForBinding](#) nor a [BindingIdentifier](#), then
 - i. **Assert**: *d* is either a [FunctionDeclaration](#), a [GeneratorDeclaration](#), an [AsyncFunctionDeclaration](#), or an [AsyncGeneratorDeclaration](#).
 - ii. NOTE: If there are multiple function declarations 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. If *varEnv* is a [global Environment Record](#), then
 - a. Let *fnDefinable* be ? *varEnv*.[CanDeclareGlobalFunction](#)(*fn*).
 - b. If *fnDefinable* is **false**, throw a **TypeError** exception.
 2. Append *fn* to *declaredFunctionNames*.
 3. Insert *d* as the first element of *functionsToInitialize*.
7. NOTE: Annex [B.3.3.3](#) adds additional steps at this point.
8. Let *declaredVarNames* be a new empty [List](#).
9. For each element *d* of *varDeclarations*, do
 - a. If *d* is a [VariableDeclaration](#), a [ForBinding](#), or a [BindingIdentifier](#), then
 - i. For each String *vn* of the [BoundNames](#) of *d*, do
 1. If *vn* is not an element of *declaredFunctionNames*, then
 - a. If *varEnv* is a [global Environment Record](#), then
 - i. Let *vnDefinable* be ? *varEnv*.[CanDeclareGlobalVar](#)(*vn*).
 - ii. If *vnDefinable* is **false**, throw a **TypeError** exception.
 - b. If *vn* is not an element of *declaredVarNames*, then
 - i. Append *vn* to *declaredVarNames*.

10. NOTE: No abnormal terminations occur after this algorithm step unless *varEnv* is a global Environment Record and the global object is a Proxy exotic object.
11. Let *lexDeclarations* be the LexicallyScopedDeclarations of *body*.
12. For each element *d* of *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. Perform ? *lexEnv*.CreateImmutableBinding(*dn*, **true**).
 - ii. Else,
 1. Perform ? *lexEnv*.CreateMutableBinding(*dn*, **false**).
13. For each Parse Node *f* of *functionsToInitialize*, do
 - a. Let *fn* be the sole element of the BoundNames of *f*.
 - b. Let *fo* be InstantiateFunctionObject of *f* with argument *lexEnv*.
 - c. If *varEnv* is a global Environment Record, then
 - i. Perform ? *varEnv*.CreateGlobalFunctionBinding(*fn*, *fo*, **true**).
 - d. Else,
 - i. Let *bindingExists* be *varEnv*.HasBinding(*fn*).
 - ii. If *bindingExists* is **false**, then
 1. Let *status* be ! *varEnv*.CreateMutableBinding(*fn*, **true**).
 2. Assert: *status* is not an abrupt completion because of validation preceding step 10.
 3. Perform ! *varEnv*.InitializeBinding(*fn*, *fo*).
 - iii. Else,
 1. Perform ! *varEnv*.SetMutableBinding(*fn*, *fo*, **false**).
14. For each String *vn* of *declaredVarNames*, do
 - a. If *varEnv* is a global Environment Record, then
 - i. Perform ? *varEnv*.CreateGlobalVarBinding(*vn*, **true**).
 - b. Else,
 - i. Let *bindingExists* be *varEnv*.HasBinding(*vn*).
 - ii. If *bindingExists* is **false**, then
 1. Let *status* be ! *varEnv*.CreateMutableBinding(*vn*, **true**).
 2. Assert: *status* is not an abrupt completion because of validation preceding step 10.
 3. Perform ! *varEnv*.InitializeBinding(*vn*, **undefined**).
15. Return NormalCompletion(empty).

NOTE An alternative version of this algorithm is described in B.3.5.

19.2.2 isFinite (*number*)

The **isFinite** function is the %isFinite% intrinsic object. When the **isFinite** function is called with one argument *number*, the following steps are taken:

1. Let *num* be ? ToNumber(*number*).
2. If *num* is NaN, +∞_F, or -∞_F, return **false**.
3. Otherwise, return **true**.

19.2.3 isNaN (*number*)

The **isNaN** function is the `%isNaN%` intrinsic object. When the **isNaN** function is called with one argument *number*, the following steps are taken:

1. Let *num* be `? ToNumber(number)`.
2. If *num* is NaN, return **true**.
3. Otherwise, return **false**.

NOTE A reliable way for ECMAScript code to test if a value **X** is a NaN is an expression of the form **X !== X**. The result will be **true** if and only if **X** is a NaN.

19.2.4 parseFloat (*string*)

The **parseFloat** function produces a **Number value** dictated by interpretation of the contents of the *string* argument as a decimal literal.

The **parseFloat** function is the `%parseFloat%` intrinsic object. When the **parseFloat** function is called with one argument *string*, the following steps are taken:

1. Let *inputString* be `? ToString(string)`.
2. Let *trimmedString* be `! TrimString(inputString, start)`.
3. If neither *trimmedString* nor any prefix of *trimmedString* satisfies the syntax of a *StrDecimalLiteral* (see 7.1.4.1), return NaN.
4. Let *numberString* be the longest prefix of *trimmedString*, which might be *trimmedString* itself, that satisfies the syntax of a *StrDecimalLiteral*.
5. Let *mathFloat* be MV of *numberString*.
6. If *mathFloat* = 0, then
 - a. If the first code unit of *trimmedString* is the code unit 0x002D (HYPHEN-MINUS), return **-0_F**.
 - b. Return **+0_F**.
7. Return **F(*mathFloat*)**.

NOTE **parseFloat** may interpret only a leading portion of *string* as a **Number value**; it ignores any code units that cannot be interpreted as part of the notation of a decimal literal, and no indication is given that any such code units were ignored.

19.2.5 parseInt (*string*, *radix*)

The **parseInt** function produces an **integral Number** dictated by interpretation of the contents of the *string* argument according to the specified *radix*. Leading white space in *string* is ignored. If *radix* is **undefined** or 0, it is assumed to be 10 except when the number begins with the code unit pairs **0x** or **0X**, in which case a radix of 16 is assumed. If *radix* is 16, the number may also optionally begin with the code unit pairs **0x** or **0X**.

The **parseInt** function is the `%parseInt%` intrinsic object. When the **parseInt** function is called, the following steps are taken:

1. Let *inputString* be `? ToString(string)`.
2. Let *S* be `! TrimString(inputString, start)`.
3. Let *sign* be 1.

4. If S is not empty and the first code unit of S is the code unit 0x002D (HYPHEN-MINUS), set $sign$ to -1.
5. If S is not empty and the first code unit of S is the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS), remove the first code unit from S .
6. Let R be $\mathbb{R}(\text{ToInt32}(\text{radix}))$.
7. Let $stripPrefix$ be **true**.
8. If $R \neq 0$, then
 - a. If $R < 2$ or $R > 36$, return **NaN**.
 - b. If $R \neq 16$, set $stripPrefix$ to **false**.
9. Else,
 - a. Set R to 10.
10. 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
 - i. Remove the first two code units from S .
 - ii. Set R to 16.
11. If S contains a code unit that is not a radix- R digit, let end be the index within S of the first such code unit; otherwise, let end be the length of S .
12. Let Z be the **substring** of S from 0 to end .
13. If Z is empty, return **NaN**.
14. Let $mathInt$ be the **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-approximated** value representing the **integer** value that is represented by Z in radix- R notation.)
15. If $mathInt = 0$, then
 - a. If $sign = -1$, return **-0_F**.
 - b. Return **+0_F**.
16. Return $\mathbb{F}(sign \times mathInt)$.

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.

19.2.6 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 19.2.6.2, 19.2.6.3, 19.2.6.4 and 19.2.6.5

NOTE

Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

19.2.6.1 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

uri :::

*uriCharacters*_{opt}

uriCharacters :::

uriCharacter *uriCharacters*_{opt}

uriCharacter :::

uriReserved

uriUnescaped

uriEscaped

uriReserved ::: **one of**

; / ? : @ & = + \$,

uriUnescaped :::

uriAlpha

DecimalDigit

uriMark

uriEscaped :::

% *HexDigit* *HexDigit*

uriAlpha ::: **one of**

a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z

uriMark ::: **one of**

- _ . ! ~ * ' ()

NOTE

The above syntax is based upon RFC 2396 and does not reflect changes introduced by the more recent RFC 3986.

Runtime Semantics

When a 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".

19.2.6.1.1 Encode (*string*, *unescapedSet*)

The abstract operation Encode takes arguments *string* (a String) and *unescapedSet* (a String). It performs URI encoding and escaping. It performs the following steps when called:

1. Let *strLen* be the number of code units in *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat,
 - a. If $k = \text{strLen}$, return *R*.
 - b. Let *C* be the code unit at index *k* within *string*.
 - c. If *C* is in *unescapedSet*, then
 - i. Set *k* to $k + 1$.
 - ii. Set *R* to the string-concatenation of *R* and *C*.
 - d. Else,
 - i. Let *cp* be !CodePointAt(*string*, *k*).
 - ii. If *cp*.[[IsUnpairedSurrogate]] is **true**, throw a **URIError** exception.
 - iii. Set *k* to $k + \text{cp}$.[[CodeUnitCount]].
 - iv. Let *Octets* be the List of octets resulting by applying the UTF-8 transformation to *cp*.
[[CodePoint]].
 - v. For each element *octet* of *Octets*, do
 1. Set *R* to the string-concatenation of:
 - *R*
 - "%"
 - the String representation of *octet*, formatted as a two-digit uppercase hexadecimal number, padded to the left with a zero if necessary

19.2.6.1.2 Decode (*string*, *reservedSet*)

The abstract operation Decode takes arguments *string* (a String) and *reservedSet* (a String). It performs URI unescaping and decoding. It performs the following steps when called:

1. Let *strLen* be the length of *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat,
 - a. If $k = \text{strLen}$, return *R*.
 - b. Let *C* be the code unit at index *k* within *string*.
 - c. If *C* is not the code unit 0x0025 (PERCENT SIGN), then
 - i. Let *S* be the String value containing only the code unit *C*.
 - d. Else,
 - i. Let *start* be *k*.
 - ii. If $k + 2 \geq \text{strLen}$, throw a **URIError** exception.
 - iii. If the code units at index (*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 index (*k* + 1) and (*k* + 2).

- v. Set k to $k + 2$.
- vi. Let n be the number of leading 1 bits in B .
- vii. If $n = 0$, then
 - 1. Let C be the code unit whose value is B .
 - 2. If C is not in *reservedSet*, then
 - a. Let S be the String value containing only the code unit C .
 - 3. Else,
 - a. Let S be the **substring** of *string* from *start* to $k + 1$.
- viii. Else,
 - 1. If $n = 1$ or $n > 4$, throw a **URIError** exception.
 - 2. If $k + (3 \times (n - 1)) \geq \text{strLen}$, throw a **URIError** exception.
 - 3. Let *Octets* be a **List** whose sole element is B .
 - 4. Let j be 1.
 - 5. Repeat, while $j < n$,
 - a. Set k to $k + 1$.
 - b. If the code unit at index k within *string* is not the code unit 0x0025 (PERCENT SIGN), throw a **URIError** exception.
 - c. If the code units at index $(k + 1)$ 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 index $(k + 1)$ and $(k + 2)$.
 - e. Set k to $k + 2$.
 - f. Append B to *Octets*.
 - g. Set j to $j + 1$.
 - 6. **Assert**: The length of *Octets* is n .
 - 7. If *Octets* does not contain a valid UTF-8 encoding of a Unicode code point, throw a **URIError** exception.
 - 8. Let V be the code point obtained by applying the UTF-8 transformation to *Octets*, that is, from a **List** of octets into a 21-bit value.
 - 9. Let S be **UTF16EncodeCodePoint**(V).
- e. Set R to the **string-concatenation** of R and S .
- f. Set k to $k + 1$.

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 49](#).

Table 49 (Informative): UTF-8 Encodings

Code Unit Value	Representation	1 st Octet	2 nd Octet	3 rd Octet	4 th Octet
0x0000 - 0x007F	00000000 0zzzzzzz	0zzzzzzz			
0x0080 - 0x07FF	00000yyy yyzzzzzz	110yyyyy	10zzzzzz		
0x0800 - 0xD7FF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	
0xD800 - 0xDBFF followed by 0xDC00 - 0xDFFF	110110vv vwwwwvwx followed by 110111yy yyzzzzzz	11110uuu	10uuwww	10xyyyyy	10zzzzzz
0xD800 - 0xDBFF not followed by 0xDC00 - 0xDFFF	causes URIError				
0xDC00 - 0xDFFF	causes URIError				
0xE000 - 0xFFFF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	

Where

$$uuuuuu = vvvv + 1$$

to account for the addition of 0x10000 as in section 3.8 of the Unicode Standard (Surrogates).

The above transformation combines each [surrogate pair](#) (for which code unit values in the inclusive range 0xD800 to 0xDFFF are reserved) into a UTF-32 representation and encodes the resulting 21-bit value into 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 0x0000. Implementations of the Decode algorithm are required to throw a **URIError** when encountering such invalid sequences.

19.2.6.2 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.

The **decodeURI** function is the *%decodeURI%* intrinsic object. When the **decodeURI** function is called with one argument *encodedURI*, the following steps are taken:

1. Let *uriString* be ? **ToString**(*encodedURI*).
2. Let *reservedURISet* be a String containing one instance of each code unit valid in *uriReserved* plus "#".
3. Return ? **Decode**(*uriString*, *reservedURISet*).

NOTE The code point # is not decoded from escape sequences even though it is not a reserved URI code point.

19.2.6.3 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.

The **decodeURIComponent** function is the *%decodeURIComponent%* intrinsic object. When the **decodeURIComponent** function is called with one argument *encodedURIComponent*, the following steps are taken:

1. Let *componentString* be ? **ToString**(*encodedURIComponent*).
2. Let *reservedURIComponentSet* be the empty String.
3. Return ? **Decode**(*componentString*, *reservedURIComponentSet*).

19.2.6.4 encodeURI (*uri*)

The **encodeURI** function computes a new version of a UTF-16 encoded (6.1.4) 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.

The **encodeURI** function is the *%encodeURI%* intrinsic object. When the **encodeURI** function is called with one argument *uri*, the following steps are taken:

1. Let *uriString* be ? **ToString**(*uri*).
2. Let *unescapedURISet* be a String containing one instance of each code unit valid in *uriReserved* and *uriUnescaped* plus "#".
3. 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.

19.2.6.5 encodeURIComponent (*uriComponent*)

The **encodeURIComponent** function computes a new version of a UTF-16 encoded (6.1.4) 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.

The **encodeURIComponent** function is the *%encodeURIComponent%* intrinsic object. When the **encodeURIComponent** function is called with one argument *uriComponent*, the following steps are taken:

1. Let *componentString* be ? [ToString\(uriComponent\)](#).
2. Let *unescapedURIComponentSet* be a String containing one instance of each code unit valid in *uriUnescaped*.
3. Return ? [Encode\(componentString, unescapedURIComponentSet\)](#).

19.3 Constructor Properties of the Global Object

19.3.1 Array (...)

See [23.1.1](#).

19.3.2 ArrayBuffer (...)

See [25.1.3](#).

19.3.3 BigInt (...)

See [21.2.1](#).

19.3.4 BigInt64Array (...)

See [23.2.5](#).

19.3.5 BigUint64Array (...)

See [23.2.5](#).

19.3.6 Boolean (...)

See [20.3.1](#).

19.3.7 DataView (...)

See [25.3.2](#).

19.3.8 Date (...)

See [21.4.2](#).

19.3.9 Error (...)

See [20.5.1](#).

19.3.10 EvalError (...)

See [20.5.5.1](#).

19.3.11 FinalizationRegistry (...)

See [26.2.1](#).

19.3.12 Float32Array (...)

See [23.2.5](#).

19.3.13 Float64Array (...)

See [23.2.5](#).

19.3.14 Function (...)

See [20.2.1](#).

19.3.15 Int8Array (...)

See [23.2.5](#).

19.3.16 Int16Array (...)

See [23.2.5](#).

19.3.17 Int32Array (...)

See [23.2.5](#).

19.3.18 Map (...)

See [24.1.1](#).

19.3.19 Number (...)

See [21.1.1](#).

19.3.20 Object (...)

See [20.1.1](#).

19.3.21 Promise (...)

See [27.2.3](#).

19.3.22 Proxy (...)

See [28.2.1](#).

19.3.23 RangeError (...)

See [20.5.5.2](#).

19.3.24 ReferenceError (...)

See [20.5.5.3](#).

19.3.25 RegExp (...)

See [22.2.3](#).

19.3.26 Set (...)

See [24.2.1](#).

19.3.27 SharedArrayBuffer (...)

See [25.2.2](#).

19.3.28 String (...)

See [22.1.1](#).

19.3.29 Symbol (...)

See [20.4.1](#).

19.3.30 SyntaxError (...)

See [20.5.5.4](#).

19.3.31 TypeError (...)

See [20.5.5.5](#).

19.3.32 Uint8Array (...)

See [23.2.5](#).

19.3.33 Uint8ClampedArray (...)

See [23.2.5](#).

19.3.34 Uint16Array (...)

See [23.2.5](#).

19.3.35 Uint32Array (...)

See [23.2.5](#).

19.3.36 URIError (...)

See [20.5.5.6](#).

19.3.37 WeakMap (...)

See [24.3.1](#).

19.3.38 WeakRef (...)

See [26.1.1](#).

19.3.39 WeakSet (...)

See [24.4](#).

19.4 Other Properties of the Global Object

19.4.1 Atomics

See [25.4](#).

19.4.2 JSON

See [25.5](#).

19.4.3 Math

See [21.3](#).

19.4.4 Reflect

See [28.1](#).

20 Fundamental Objects

20.1 Object Objects

20.1.1 The Object Constructor

The Object **constructor**:

- is *%Object%*.
- is the initial value of the "Object" property of the [global object](#).
- creates a new [ordinary object](#) when called as a **constructor**.
- performs a type conversion when called as a function rather than as a **constructor**.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition.

20.1.1.1 Object ([*value*])

When the **Object** function is called with optional argument *value*, the following steps are taken:

1. If NewTarget is neither **undefined** nor the active function, then
 - a. Return ? [OrdinaryCreateFromConstructor](#)(NewTarget, "%Object.prototype%").
2. If *value* is **undefined** or **null**, return ! [OrdinaryObjectCreate](#)(%Object.prototype%).
3. Return ! [ToObject](#)(*value*).

The "length" property of the **Object** function is 1_F.

20.1.2 Properties of the Object Constructor

The Object **constructor**:

- has a [\[\[Prototype\]\]](#) internal slot whose value is *%Function.prototype%*.
- has a "length" property.
- has the following additional properties:

20.1.2.1 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. If only one argument was passed, return *to*.
3. For each element *nextSource* of *sources*, do
 - a. If *nextSource* is neither **undefined** nor **null**, then
 - i. Let *from* be ! [ToObject](#)(*nextSource*).
 - ii. Let *keys* be ? *from*.[\[\[OwnPropertyKeys\]\]](#)()
 - iii. For each element *nextKey* of *keys*, do
 1. Let *desc* be ? *from*.[\[\[GetOwnProperty\]\]](#)(*nextKey*).
 2. If *desc* is not **undefined** and *desc*.[\[\[Enumerable\]\]](#) is **true**, then
 - a. Let *propValue* be ? [Get](#)(*from*, *nextKey*).
 - b. Perform ? [Set](#)(*to*, *nextKey*, *propValue*, **true**).
4. Return *to*.

The "length" property of the **assign** function is 2.

20.1.2.2 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 neither Object nor Null, throw a **TypeError** exception.
2. Let *obj* be ! **OrdinaryObjectCreate**(*O*).
3. If *Properties* is not **undefined**, then
 - a. Return ? **ObjectDefineProperties**(*obj*, *Properties*).
4. Return *obj*.

20.1.2.3 Object.defineProperties (*O*, *Properties*)

The **defineProperties** function is used to add own properties and/or update the attributes of existing own properties of an object. When the **defineProperties** function is called, the following steps are taken:

1. If **Type**(*O*) is not Object, throw a **TypeError** exception.
2. Return ? **ObjectDefineProperties**(*O*, *Properties*).

20.1.2.3.1 ObjectDefineProperties (*O*, *Properties*)

The abstract operation **ObjectDefineProperties** takes arguments *O* and *Properties*. It performs the following steps when called:

1. **Assert**: **Type**(*O*) is Object.
2. Let *props* be ? **ToObject**(*Properties*).
3. Let *keys* be ? *props*.[[OwnPropertyKeys]]().
4. Let *descriptors* be a new empty **List**.
5. For each element *nextKey* of *keys*, do
 - a. Let *propDesc* be ? *props*.[[GetOwnProperty]](*nextKey*).
 - b. If *propDesc* is not **undefined** and *propDesc*.[[Enumerable]] is **true**, then
 - i. Let *descObj* be ? **Get**(*props*, *nextKey*).
 - ii. Let *desc* be ? **ToPropertyDescriptor**(*descObj*).
 - iii. Append the pair (a two element **List**) consisting of *nextKey* and *desc* to the end of *descriptors*.
6. For each element *pair* of *descriptors*, do
 - a. Let *P* be the first element of *pair*.
 - b. Let *desc* be the second element of *pair*.
 - c. Perform ? **DefinePropertyOrThrow**(*O*, *P*, *desc*).
7. Return *O*.

20.1.2.4 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. Let *desc* be ? [ToPropertyDescriptor](#)(*Attributes*).
4. Perform ? [DefinePropertyOrThrow](#)(*O*, *key*, *desc*).
5. Return *O*.

20.1.2.5 [Object.entries](#) (*O*)

When the **entries** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ? [ToObject](#)(*O*).
2. Let *nameList* be ? [EnumerableOwnPropertyNames](#)(*obj*, key+value).
3. Return [CreateArrayFromList](#)(*nameList*).

20.1.2.6 [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 ? [SetIntegrityLevel](#)(*O*, frozen).
3. If *status* is **false**, throw a **TypeError** exception.
4. Return *O*.

20.1.2.7 [Object.fromEntries](#) (*iterable*)

When the **fromEntries** method is called with argument *iterable*, the following steps are taken:

1. Perform ? [RequireObjectCoercible](#)(*iterable*).
2. Let *obj* be ! [OrdinaryObjectCreate](#)(%[Object.prototype](#)%).
3. **Assert**: *obj* is an extensible [ordinary object](#) with no own properties.
4. Let *stepsDefine* be the algorithm steps defined in [CreateDataPropertyOnObject Functions](#).
5. Let *lengthDefine* be the number of non-optional parameters of the function definition in [CreateDataPropertyOnObject Functions](#).
6. Let *adder* be ! [CreateBuiltinFunction](#)(*stepsDefine*, *lengthDefine*, "", « »).
7. Return ? [AddEntriesFromIterable](#)(*obj*, *iterable*, *adder*).

NOTE The function created for *adder* is never directly accessible to ECMAScript code.

20.1.2.7.1 [CreateDataPropertyOnObject Functions](#)

A [CreateDataPropertyOnObject](#) function is an anonymous built-in function. When a [CreateDataPropertyOnObject](#) function is called with arguments *key* and *value*, the following steps are taken:

1. Let *O* be the **this** value.
2. **Assert**: [Type](#)(*O*) is Object.
3. **Assert**: *O* is an extensible [ordinary object](#).
4. Let *propertyKey* be ? [ToPropertyKey](#)(*key*).
5. Perform ! [CreateDataPropertyOrThrow](#)(*O*, *propertyKey*, *value*).
6. Return **undefined**.

20.1.2.8 Object.getOwnPropertyDescriptor (*O*, *P*)

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

1. Let *obj* be ? [ToObject](#)(*O*).
2. Let *key* be ? [ToPropertyKey](#)(*P*).
3. Let *desc* be ? *obj*.[[GetOwnProperty]](*key*).
4. Return [FromPropertyDescriptor](#)(*desc*).

20.1.2.9 Object.getOwnPropertyDescriptors (*O*)

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

1. Let *obj* be ? [ToObject](#)(*O*).
2. Let *ownKeys* be ? *obj*.[[OwnPropertyKeys]]().
3. Let *descriptors* be ! [OrdinaryObjectCreate](#)(%Object.prototype%).
4. For each element *key* of *ownKeys*, do
 - a. Let *desc* be ? *obj*.[[GetOwnProperty]](*key*).
 - b. Let *descriptor* be ! [FromPropertyDescriptor](#)(*desc*).
 - c. If *descriptor* is not **undefined**, perform ! [CreateDataPropertyOrThrow](#)(*descriptors*, *key*, *descriptor*).
5. Return *descriptors*.

20.1.2.10 Object.getOwnPropertyNames (*O*)

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

1. Return ? [GetOwnPropertyKeys](#)(*O*, string).

20.1.2.11 Object.getOwnPropertySymbols (*O*)

When the **getOwnPropertySymbols** function is called with argument *O*, the following steps are taken:

1. Return ? [GetOwnPropertyKeys](#)(*O*, symbol).

20.1.2.11.1 GetOwnPropertyKeys (*O*, *type*)

The abstract operation [GetOwnPropertyKeys](#) takes arguments *O* and *type* (either string or symbol). It performs the following steps when called:

1. Let *obj* be ? [ToObject](#)(*O*).
2. Let *keys* be ? *obj*.[[OwnPropertyKeys]]().
3. Let *nameList* be a new empty [List](#).
4. For each element *nextKey* of *keys*, do
 - a. If [Type](#)(*nextKey*) is Symbol and *type* is symbol or [Type](#)(*nextKey*) is String and *type* is string, then
 - i. Append *nextKey* as the last element of *nameList*.
5. Return [CreateArrayFromList](#)(*nameList*).

20.1.2.12 Object.getPrototypeOf (*O*)

When the **getPrototypeOf** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ? [ToObject](#)(*O*).
2. Return ? *obj*.[[GetPrototypeOf]]().

20.1.2.13 Object.is (*value1*, *value2*)

When the **is** function is called with arguments *value1* and *value2*, the following steps are taken:

1. Return [SameValue](#)(*value1*, *value2*).

20.1.2.14 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 ? [IsExtensible](#)(*O*).

20.1.2.15 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).

20.1.2.16 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).

20.1.2.17 Object.keys (*O*)

When the **keys** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ? [ToObject](#)(*O*).
2. Let *nameList* be ? [EnumerableOwnPropertyNames](#)(*obj*, key).
3. Return [CreateArrayFromList](#)(*nameList*).

20.1.2.18 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 ? *O*.[[PreventExtensions]]().
3. If *status* is **false**, throw a **TypeError** exception.
4. Return *O*.

20.1.2.19 Object.prototype

The initial value of **Object.prototype** is the [Object prototype object](#).

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

20.1.2.20 `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 `? SetIntegrityLevel(O, sealed)`.
3. If `status` is `false`, throw a `TypeError` exception.
4. Return `O`.

20.1.2.21 `Object.setPrototypeOf (O, proto)`

When the **setPrototypeOf** function is called with arguments `O` and `proto`, the following steps are taken:

1. Set `O` to `? RequireObjectCoercible(O)`.
2. If `Type(proto)` is neither `Object` nor `Null`, throw a `TypeError` exception.
3. If `Type(O)` is not `Object`, return `O`.
4. Let `status` be `? O.[[SetPrototypeOf]](proto)`.
5. If `status` is `false`, throw a `TypeError` exception.
6. Return `O`.

20.1.2.22 `Object.values (O)`

When the **values** function is called with argument `O`, the following steps are taken:

1. Let `obj` be `? ToObject(O)`.
2. Let `nameList` be `? EnumerableOwnPropertyNames(obj, value)`.
3. Return `CreateArrayFromList(nameList)`.

20.1.3 Properties of the Object Prototype Object

The *Object prototype object*:

- is `%Object.prototype%`.
- has an `[[Extensible]]` internal slot whose value is `true`.
- has the internal methods defined for ordinary objects, except for the `[[SetPrototypeOf]]` method, which is as defined in 10.4.7.1. (Thus, it is an `immutable prototype exotic object`.)
- has a `[[Prototype]]` internal slot whose value is `null`.

20.1.3.1 `Object.prototype.constructor`

The initial value of `Object.prototype.constructor` is `%Object%`.

20.1.3.2 `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. Let O be ? `ToObject(this value)`.
3. Return ? `HasOwnProperty(O, P)`.

NOTE The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the **this** value is **undefined** or **null**.

20.1.3.3 `Object.prototype.isPrototypeOf (V)`

When the **isPrototypeOf** method is called with argument V , the following steps are taken:

1. If `Type(V)` is not `Object`, return **false**.
2. Let O be ? `ToObject(this value)`.
3. Repeat,
 - a. Set V to ? V .[[`GetPrototypeOf`]]().
 - b. If V is **null**, return **false**.
 - c. If `SameValue(O, V)` is **true**, 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**.

20.1.3.4 `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. Let O be ? `ToObject(this value)`.
3. Let $desc$ be ? O .[[`GetOwnProperty`]](P).
4. If $desc$ is **undefined**, return **false**.
5. Return $desc$.[[`Enumerable`]].

NOTE 1 This method does not consider objects in the prototype chain.

NOTE 2 The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the **this** value is **undefined** or **null**.

20.1.3.5 `Object.prototype.toLocaleString ([reserved1 [, reserved2]])`

When the **toLocaleString** method is called, the following steps are taken:

1. Let O be the **this** value.
2. Return ? `Invoke(O, "toString")`.

The optional parameters to this function are not used 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 `%TypedArray%` provide their own locale-sensitive **toLocaleString** methods.

NOTE 2 ECMA-402 intentionally does not provide an alternative to this default implementation.

20.1.3.6 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 ! **ToObject**(**this** value).
4. Let *isArray* be ? **isArray**(*O*).
5. If *isArray* is **true**, let *builtinTag* be "Array".
6. Else if *O* has a [[ParameterMap]] internal slot, let *builtinTag* be "Arguments".
7. Else if *O* has a [[Call]] internal method, let *builtinTag* be "Function".
8. Else if *O* has an [[ErrorData]] internal slot, let *builtinTag* be "Error".
9. Else if *O* has a [[BooleanData]] internal slot, let *builtinTag* be "Boolean".
10. Else if *O* has a [[NumberData]] internal slot, let *builtinTag* be "Number".
11. Else if *O* has a [[StringData]] internal slot, let *builtinTag* be "String".
12. Else if *O* has a [[DateValue]] internal slot, let *builtinTag* be "Date".
13. Else if *O* has a [[RegExpMatcher]] internal slot, let *builtinTag* be "RegExp".
14. Else, let *builtinTag* be "Object".
15. Let *tag* be ? **Get**(*O*, @@**toStringTag**).
16. If **Type**(*tag*) is not String, set *tag* to *builtinTag*.
17. Return the **string-concatenation** of "[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 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.

20.1.3.7 Object.prototype.valueOf ()

When the **valueOf** method is called, the following steps are taken:

1. Return ? **ToObject**(**this** value).

20.1.4 Properties of Object Instances

Object instances have no special properties beyond those inherited from the [Object prototype object](#).

20.2 Function Objects

20.2.1 The Function Constructor

The Function [constructor](#):

- is `%Function%`.
- is the initial value of the "Function" property of the [global object](#).
- creates and initializes a new [function object](#) when called as a function rather than as a [constructor](#). Thus the function call `Function(…)` is equivalent to the object creation expression `new Function(…)` with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified Function behaviour must include a **super** call to the Function [constructor](#) to create and initialize a subclass instance with the internal slots necessary for built-in function behaviour. All ECMAScript 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 `GeneratorFunction`, `AsyncFunction`, and `AsyncGeneratorFunction` subclasses.

20.2.1.1 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 *C* be the [active function object](#).
2. Let *args* be the [argumentsList](#) that was passed to this function by `[[Call]]` or `[[Construct]]`.
3. Return ? `CreateDynamicFunction(C, NewTarget, normal, args)`.

NOTE

It is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
new Function("a", "b", "c", "return a+b+c")
new Function("a, b, c", "return a+b+c")
new Function("a,b", "c", "return a+b+c")
```

20.2.1.1.1 CreateDynamicFunction (*constructor, newTarget, kind, args*)

The abstract operation `CreateDynamicFunction` takes arguments *constructor* (a [constructor](#)), *newTarget* (a [constructor](#)), *kind* (either `normal`, `generator`, `async`, or `asyncGenerator`), and *args* (a [List](#) of ECMAScript language values). *constructor* is the [constructor](#) function that is performing this action. *newTarget* is the [constructor](#) that **new** was initially applied to. *args* is the argument values that were passed to *constructor*. It performs the following steps when called:

1. **Assert**: The [execution context stack](#) has at least two elements.

2. Let *callerContext* be the second to top element of the [execution context stack](#).
3. Let *callerRealm* be *callerContext*'s [Realm](#).
4. Let *calleeRealm* be the [current Realm Record](#).
5. Perform ? [HostEnsureCanCompileStrings](#)(*callerRealm*, *calleeRealm*).
6. If *newTarget* is **undefined**, set *newTarget* to *constructor*.
7. If *kind* is **normal**, then
 - a. Let *goal* be the grammar symbol *FunctionBody*_[~Yield, ~Await].
 - b. Let *parameterGoal* be the grammar symbol *FormalParameters*_[~Yield, ~Await].
 - c. Let *fallbackProto* be "%**Function.prototype**%".
8. Else if *kind* is **generator**, then
 - a. Let *goal* be the grammar symbol *GeneratorBody*.
 - b. Let *parameterGoal* be the grammar symbol *FormalParameters*_[+Yield, ~Await].
 - c. Let *fallbackProto* be "%**GeneratorFunction.prototype**%".
9. Else if *kind* is **async**, then
 - a. Let *goal* be the grammar symbol *AsyncFunctionBody*.
 - b. Let *parameterGoal* be the grammar symbol *FormalParameters*_[~Yield, +Await].
 - c. Let *fallbackProto* be "%**AsyncFunction.prototype**%".
10. Else,
 - a. **Assert**: *kind* is **asyncGenerator**.
 - b. Let *goal* be the grammar symbol *AsyncGeneratorBody*.
 - c. Let *parameterGoal* be the grammar symbol *FormalParameters*_[+Yield, +Await].
 - d. Let *fallbackProto* be "%**AsyncGeneratorFunction.prototype**%".
11. Let *argCount* be the number of elements in *args*.
12. Let *P* be the empty String.
13. If *argCount* = 0, let *bodyArg* be the empty String.
14. Else if *argCount* = 1, let *bodyArg* be *args*[0].
15. Else,
 - a. **Assert**: *argCount* > 1.
 - b. Let *firstArg* be *args*[0].
 - c. Set *P* to ? [ToString](#)(*firstArg*).
 - d. Let *k* be 1.
 - e. Repeat, while *k* < *argCount* - 1,
 - i. Let *nextArg* be *args*[*k*].
 - ii. Let *nextArgString* be ? [ToString](#)(*nextArg*).
 - iii. Set *P* to the [string-concatenation](#) of *P*, ",", (a comma), and *nextArgString*.
 - iv. Set *k* to *k* + 1.
 - f. Let *bodyArg* be *args*[*k*].
16. Let *bodyString* be the [string-concatenation](#) of 0x000A (LINE FEED), ? [ToString](#)(*bodyArg*), and 0x000A (LINE FEED).
17. Let *prefix* be the prefix associated with *kind* in [Table 50](#).
18. Let *sourceString* be the [string-concatenation](#) of *prefix*, " **anonymous**(" , *P*, 0x000A (LINE FEED), ")" {" , *bodyString*, and "}".
19. Let *sourceText* be ! [StringToCodePoints](#)(*sourceString*).
20. Perform the following substeps in an [implementation-defined](#) order, possibly interleaving parsing and error detection:
 - a. Let *parameters* be [ParseText](#)(! [StringToCodePoints](#)(*P*), *parameterGoal*).
 - b. If *parameters* is a [List](#) of errors, throw a **SyntaxError** exception.

- c. Let *body* be `ParseText(! StringToCodePoints(bodyString), goal)`.
- d. If *body* is a List of errors, throw a **SyntaxError** exception.
- e. Let *strict* be `FunctionBodyContainsUseStrict` of *body*.
- f. If *strict* is **true**, apply the early error rules for `UniqueFormalParameters : FormalParameters` to *parameters*.
- g. If *strict* is **true** and `IsSimpleParameterList` of *parameters* is **false**, throw a **SyntaxError** exception.
- h. If any element of the `BoundNames` of *parameters* also occurs in the `LexicallyDeclaredNames` of *body*, throw a **SyntaxError** exception.
- i. If *body* `Contains SuperCall` is **true**, throw a **SyntaxError** exception.
- j. If *parameters* `Contains SuperCall` is **true**, throw a **SyntaxError** exception.
- k. If *body* `Contains SuperProperty` is **true**, throw a **SyntaxError** exception.
- l. If *parameters* `Contains SuperProperty` is **true**, throw a **SyntaxError** exception.
- m. If *kind* is generator or `asyncGenerator`, then
 - i. If *parameters* `Contains YieldExpression` is **true**, throw a **SyntaxError** exception.
- n. If *kind* is `async` or `asyncGenerator`, then
 - i. If *parameters* `Contains AwaitExpression` is **true**, throw a **SyntaxError** exception.
- o. If *strict* is **true**, then
 - i. If `BoundNames` of *parameters* contains any duplicate elements, throw a **SyntaxError** exception.
- 21. Let *proto* be `? GetPrototypeFromConstructor(newTarget, fallbackProto)`.
- 22. Let *realmF* be the current Realm Record.
- 23. Let *scope* be `realmF.[[GlobalEnv]]`.
- 24. Let *F* be `! OrdinaryFunctionCreate(proto, sourceText, parameters, body, non-lexical-this, scope)`.
- 25. Perform `SetFunctionName(F, "anonymous")`.
- 26. If *kind* is generator, then
 - a. Let *prototype* be `! OrdinaryObjectCreate(%GeneratorFunction.prototype.prototype%)`.
 - b. Perform `DefinePropertyOrThrow(F, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.
- 27. Else if *kind* is `asyncGenerator`, then
 - a. Let *prototype* be `! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.prototype%)`.
 - b. Perform `DefinePropertyOrThrow(F, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.
- 28. Else if *kind* is normal, perform `MakeConstructor(F)`.
- 29. NOTE: Functions whose *kind* is `async` are not constructible and do not have a `[[Construct]]` internal method or a **"prototype"** property.
- 30. Return *F*.

NOTE

CreateDynamicFunction defines a **"prototype"** property on any function it creates whose *kind* is not `async` to provide for the possibility that the function will be used as a **constructor**.

Table 50: Dynamic Function SourceText Prefixes

Kind	Prefix
normal	"function"
generator	"function**"
async	"async function"
asyncGenerator	"async function**"

20.2.2 Properties of the Function Constructor

The Function `constructor`:

- is itself a built-in `function object`.
- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

20.2.2.1 `Function.length`

This is a `data property` with a value of 1. This property has the attributes { `[[Writable]]: false`, `[[Enumerable]]: false`, `[[Configurable]]: true` }.

20.2.2.2 `Function.prototype`

The value of `Function.prototype` is the `Function prototype object`.

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

20.2.3 Properties of the Function Prototype Object

The *Function prototype object*:

- is `%Function.prototype%`.
- is itself a built-in `function object`.
- accepts any arguments and returns `undefined` when invoked.
- does not have a `[[Construct]]` internal method; it cannot be used as a `constructor` with the `new` operator.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- does not have a `"prototype"` property.
- has a `"length"` property whose value is `+0`.
- has a `"name"` property whose value is the empty String.

NOTE The Function prototype object is specified to be a `function object` to ensure compatibility with ECMAScript code that was created prior to the ECMAScript 2015 specification.

20.2.3.1 `Function.prototype.apply (thisArg, argArray)`

When the `apply` method is called with arguments `thisArg` and `argArray`, the following steps are taken:

1. Let `func` be the `this` value.
2. If `IsCallable(func)` is `false`, throw a `TypeError` exception.
3. If `argArray` is `undefined` or `null`, then
 - a. Perform `PrepareForTailCall()`.
 - b. Return `? Call(func, thisArg)`.
4. Let `argList` be `? CreateListFromArrayLike(argArray)`.
5. Perform `PrepareForTailCall()`.
6. Return `? Call(func, thisArg, argList)`.

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 functions still perform these transformations upon entry to the function.

NOTE 2 If *func* is an arrow function or a **bound function exotic object** then the *thisArg* will be ignored by the function **[[Call]]** in step 6.

20.2.3.2 **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 *F* be ? **BoundFunctionCreate**(*Target*, *thisArg*, *args*).
4. Let *L* be 0.
5. Let *targetHasLength* be ? **HasOwnProperty**(*Target*, "length").
6. If *targetHasLength* is **true**, then
 - a. Let *targetLen* be ? **Get**(*Target*, "length").
 - b. If **Type**(*targetLen*) is **Number**, then
 - i. If *targetLen* is $+\infty_{\mathbb{F}}$, set *L* to $+\infty$.
 - ii. Else if *targetLen* is $-\infty_{\mathbb{F}}$, set *L* to 0.
 - iii. Else,
 1. Let *targetLenAsInt* be ! **ToIntegerOrInfinity**(*targetLen*).
 2. **Assert**: *targetLenAsInt* is finite.
 3. Let *argCount* be the number of elements in *args*.
 4. Set *L* to **max**(*targetLenAsInt* - *argCount*, 0).
7. Perform ! **SetFunctionLength**(*F*, *L*).
8. Let *targetName* be ? **Get**(*Target*, "name").
9. If **Type**(*targetName*) is not **String**, set *targetName* to the empty **String**.
10. Perform **SetFunctionName**(*F*, *targetName*, "bound").
11. Return *F*.

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 function exotic object** then the *thisArg* passed to this method will not be used by subsequent calls to *F*.

20.2.3.3 **Function.prototype.call** (*thisArg*, ...*args*)

When the **call** method is called with argument *thisArg* and zero or more *args*, the following steps are taken:

1. Let *func* be the **this** value.

2. If `IsCallable(func)` is **false**, throw a **TypeError** exception.
3. Perform `PrepareForTailCall()`.
4. Return `? Call(func, thisArg, args)`.

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 functions still perform these transformations upon entry to the function.

NOTE 2 If *func* is an arrow function or a **bound function exotic object** then the *thisArg* will be ignored by the function `[[Call]]` in step 4.

20.2.3.4 `Function.prototype.constructor`

The initial value of **`Function.prototype.constructor`** is `%Function%`.

20.2.3.5 `Function.prototype.toString()`

When the **`toString`** method is called, the following steps are taken:

1. Let *func* be the **this** value.
2. If `Type(func)` is **Object** and *func* has a `[[SourceText]]` internal slot and *func*.`[[SourceText]]` is a sequence of Unicode code points and `! HostHasSourceTextAvailable(func)` is **true**, then
 - a. Return `! CodePointsToString(func. [[SourceText]])`.
3. If *func* is a **built-in function object**, return an **implementation-defined** String source code representation of *func*. The representation must have the syntax of a *NativeFunction*. Additionally, if *func* has an `[[InitialName]]` internal slot and *func*.`[[InitialName]]` is a String, the portion of the returned String that would be matched by `NativeFunctionAccessoropt PropertyName` must be the value of *func*.`[[InitialName]]`.
4. If `Type(func)` is **Object** and `IsCallable(func)` is **true**, return an **implementation-defined** String source code representation of *func*. The representation must have the syntax of a *NativeFunction*.
5. Throw a **TypeError** exception.

NativeFunction :

```
function NativeFunctionAccessoropt PropertyName[~Yield, ~Await] opt (
  FormalParameters[~Yield, ~Await] ) { [ native code ] }
```

NativeFunctionAccessor :

```
get
set
```

20.2.3.6 `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 `? OrdinaryHasInstance(F, V)`.

The value of the "name" property of this function is "[Symbol.hasInstance]".

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

NOTE

This is the default implementation of **@@hasInstance** that most functions inherit. **@@hasInstance** is called by the **instanceof** operator to determine whether a value is an instance of a specific **constructor**. An expression such as

```
v 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.

20.2.4 Function Instances

Every Function instance is an ECMAScript **function object** and has the internal slots listed in [Table 29](#). Function objects created using the **Function.prototype.bind** method ([20.2.3.2](#)) have the internal slots listed in [Table 30](#).

Function instances have the following properties:

20.2.4.1 length

The value of the "length" property is an **integral Number** 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 }.

20.2.4.2 name

The value of the "name" property is a String that is descriptive of the function. The name has no semantic significance but is typically a variable or **property name** that is used to refer to the function at its point of definition in 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 use the empty String as the value of the "name" property.

20.2.4.3 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 the object created when that function is invoked as a **constructor**.

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 *AsyncGeneratorMethod*) or an *ArrowFunction* do not have a "prototype" property.

20.2.5 HostHasSourceTextAvailable (*func*)

The *host-defined* abstract operation `HostHasSourceTextAvailable` takes argument *func* (a *function object*). It allows *host* environments to prevent the source text from being provided for *func*.

An implementation of `HostHasSourceTextAvailable` must complete normally in all cases. This operation must be deterministic with respect to its parameters. Each time it is called with a specific *func* as its argument, it must return the same completion record. The default implementation of `HostHasSourceTextAvailable` is to unconditionally return a normal completion with a value of `true`.

20.3 Boolean Objects

20.3.1 The Boolean Constructor

The Boolean *constructor*:

- is *%Boolean%*.
- is the initial value of the "Boolean" property of the *global object*.
- creates and initializes a new Boolean object when called as a *constructor*.
- performs a type conversion when called as a function rather than as a *constructor*.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified Boolean behaviour must include a **super** call to the Boolean *constructor* to create and initialize the subclass instance with a `[[BooleanData]]` internal slot.

20.3.1.1 Boolean (*value*)

When **Boolean** is called with argument *value*, the following steps are taken:

1. Let *b* be ! `ToBoolean(value)`.
2. If `NewTarget` is **undefined**, return *b*.
3. Let *O* be ? `OrdinaryCreateFromConstructor(NewTarget, "%Boolean.prototype%", « [[BooleanData]] »)`.
4. Set *O*.`[[BooleanData]]` to *b*.
5. Return *O*.

20.3.2 Properties of the Boolean Constructor

The Boolean *constructor*:

- has a `[[Prototype]]` internal slot whose value is *%Function.prototype%*.
- has the following properties:

20.3.2.1 Boolean.prototype

The initial value of **Boolean.prototype** is the [Boolean prototype object](#).

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

20.3.3 Properties of the Boolean Prototype Object

The *Boolean prototype object*:

- is *%Boolean.prototype%*.
- is an [ordinary object](#).
- is itself a Boolean object; it has a [[BooleanData]] internal slot with the value **false**.
- has a [[Prototype]] internal slot whose value is *%Object.prototype%*.

The abstract operation *thisBooleanValue* takes argument *value*. It performs the following steps when called:

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 *value*.[[BooleanData]].
 - b. **Assert**: **Type**(*b*) is Boolean.
 - c. Return *b*.
3. Throw a **TypeError** exception.

20.3.3.1 Boolean.prototype.constructor

The initial value of **Boolean.prototype.constructor** is *%Boolean%*.

20.3.3.2 Boolean.prototype.toString ()

The following steps are taken:

1. Let *b* be ? **thisBooleanValue**(**this** value).
2. If *b* is **true**, return **"true"**; else return **"false"**.

20.3.3.3 Boolean.prototype.valueOf ()

The following steps are taken:

1. Return ? **thisBooleanValue**(**this** value).

20.3.4 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.

20.4 Symbol Objects

20.4.1 The Symbol Constructor

The Symbol [constructor](#):

- is `%Symbol%`.
- is the initial value of the "Symbol" property of the [global object](#).
- returns a new Symbol value when called as a function.
- is not intended to be used with the **new** operator.
- is not intended to be subclassed.
- may be used as the value of an **extends** clause of a class definition but a **super** call to it will cause an exception.

20.4.1.1 Symbol ([*description*])

When **Symbol** is called with optional argument *description*, the following steps are taken:

1. If `NewTarget` is not **undefined**, throw a **TypeError** exception.
2. If *description* is **undefined**, let *descString* be **undefined**.
3. Else, let *descString* be `? ToString(description)`.
4. Return a new unique Symbol value whose `[[Description]]` value is *descString*.

20.4.2 Properties of the Symbol Constructor

The Symbol [constructor](#):

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

20.4.2.1 Symbol.asyncIterator

The initial value of **Symbol.asyncIterator** is the well known symbol `@@asyncIterator` ([Table 1](#)).

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

20.4.2.2 Symbol.for (*key*)

When **Symbol.for** is called with argument *key* it performs the following steps:

1. Let *stringKey* be `? ToString(key)`.
2. For each element *e* of the `GlobalSymbolRegistry` [List](#), do
 - a. If `SameValue(e.[[Key]], stringKey)` is **true**, return `e.[[Symbol]]`.
3. **Assert**: `GlobalSymbolRegistry` does not currently contain an entry for *stringKey*.
4. Let *newSymbol* be a new unique Symbol value whose `[[Description]]` value is *stringKey*.
5. Append the [Record](#) { `[[Key]]`: *stringKey*, `[[Symbol]]`: *newSymbol* } to the `GlobalSymbolRegistry` [List](#).
6. Return *newSymbol*.

The `GlobalSymbolRegistry` is a [List](#) that is globally available. It is shared by all realms. Prior to the evaluation of any ECMAScript code it is initialized as a new empty [List](#). Elements of the `GlobalSymbolRegistry` are [Records](#) with the structure defined in [Table 51](#).

Table 51: GlobalSymbolRegistry Record Fields

Field Name	Value	Usage
[[Key]]	A String	A string key used to globally identify a Symbol.
[[Symbol]]	A Symbol	A symbol that can be retrieved from any realm .

20.4.2.3 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** }.

20.4.2.4 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** }.

20.4.2.5 Symbol.iterator

The initial value of **Symbol.iterator** is the well-known symbol [@@iterator](#) (Table 1).

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

20.4.2.6 Symbol.keyFor (*sym*)

When **Symbol.keyFor** is called with argument *sym* it performs the following steps:

1. If **Type**(*sym*) is not Symbol, throw a **TypeError** exception.
2. For each element *e* of the GlobalSymbolRegistry List (see 20.4.2.2), do
 - a. If **SameValue**(*e*.[[Symbol]], *sym*) is **true**, return *e*.[[Key]].
3. **Assert**: GlobalSymbolRegistry does not currently contain an entry for *sym*.
4. Return **undefined**.

20.4.2.7 Symbol.match

The initial value of **Symbol.match** is the well-known symbol [@@match](#) (Table 1).

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

20.4.2.8 Symbol.matchAll

The initial value of **Symbol.matchAll** is the well-known symbol [@@matchAll](#) (Table 1).

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

20.4.2.9 Symbol.prototype

The initial value of **Symbol.prototype** is the [Symbol prototype object](#).

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

20.4.2.10 `Symbol.replace`

The initial value of `Symbol.replace` is the well-known symbol `@@replace` (Table 1).

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

20.4.2.11 `Symbol.search`

The initial value of `Symbol.search` is the well-known symbol `@@search` (Table 1).

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

20.4.2.12 `Symbol.species`

The initial value of `Symbol.species` is the well-known symbol `@@species` (Table 1).

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

20.4.2.13 `Symbol.split`

The initial value of `Symbol.split` is the well-known symbol `@@split` (Table 1).

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

20.4.2.14 `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` }.

20.4.2.15 `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` }.

20.4.2.16 `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` }.

20.4.3 Properties of the Symbol Prototype Object

The *Symbol prototype object*:

- is `%Symbol.prototype%`.
- is an [ordinary object](#).

- is not a Symbol instance and does not have a `[[SymbolData]]` internal slot.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.

The abstract operation `thisSymbolValue` takes argument `value`. It performs the following steps when called:

1. If `Type(value)` is Symbol, return `value`.
2. If `Type(value)` is Object and `value` has a `[[SymbolData]]` internal slot, then
 - a. Let `s` be `value.[[SymbolData]]`.
 - b. **Assert:** `Type(s)` is Symbol.
 - c. Return `s`.
3. Throw a **TypeError** exception.

20.4.3.1 Symbol.prototype.constructor

The initial value of `Symbol.prototype.constructor` is `%Symbol%`.

20.4.3.2 get Symbol.prototype.description

`Symbol.prototype.description` 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. Let `sym` be ? `thisSymbolValue(s)`.
3. Return `sym.[[Description]]`.

20.4.3.3 Symbol.prototype.toString ()

The following steps are taken:

1. Let `sym` be ? `thisSymbolValue(this value)`.
2. Return `SymbolDescriptiveString(sym)`.

20.4.3.3.1 SymbolDescriptiveString (sym)

The abstract operation `SymbolDescriptiveString` takes argument `sym`. It performs the following steps when called:

1. **Assert:** `Type(sym)` is Symbol.
2. Let `desc` be `sym`'s `[[Description]]` value.
3. If `desc` is **undefined**, set `desc` to the empty String.
4. **Assert:** `Type(desc)` is String.
5. Return the **string-concatenation** of `"Symbol(", desc, and "`".

20.4.3.4 Symbol.prototype.valueOf ()

The following steps are taken:

1. Return ? `thisSymbolValue(this value)`.

20.4.3.5 Symbol.prototype [@@toPrimitive] (hint)

This function is called by ECMAScript language operators to convert a Symbol object to a primitive value.

When the **@@toPrimitive** method is called with argument *hint*, the following steps are taken:

1. Return ? **thisSymbolValue**(**this** value).

The value of the "name" property of this function is "[Symbol.toPrimitive]".

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

NOTE The argument is ignored.

20.4.3.6 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** }.

20.4.4 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.

20.5 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.

When an ECMAScript implementation detects a runtime error, it throws a new instance of one of the *NativeError* objects defined in 20.5.5 or a new instance of AggregateError object defined in 20.5.7. 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, in the **implementation-defined message** property of the prototype object, and in the presence of the *%AggregateError%*-specific **errors** property.

20.5.1 The Error Constructor

The Error **constructor**:

- is *%Error%*.
- is the initial value of the "Error" property of the **global object**.
- creates and initializes a new Error object when called as a function rather than as a **constructor**. Thus the function call **Error(...)** is equivalent to the object creation expression **new Error(...)** with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified Error behaviour must include a **super** call to the Error **constructor** to create and initialize subclass instances with an [[ErrorData]] internal slot.

20.5.1.1 Error (*message*)

When the **Error** function is called with argument *message*, the following steps are taken:

1. If *NewTarget* is **undefined**, let *newTarget* be the **active function object**; else let *newTarget* be *NewTarget*.
2. Let *O* be ? **OrdinaryCreateFromConstructor**(*newTarget*, "%Error.prototype%", « [[ErrorData]] »).
3. 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. Perform ! **DefinePropertyOrThrow**(*O*, "message", *msgDesc*).
4. Return *O*.

20.5.2 Properties of the Error Constructor

The Error **constructor**:

- has a [[Prototype]] internal slot whose value is %Function.prototype%.
- has the following properties:

20.5.2.1 Error.prototype

The initial value of **Error.prototype** is the **Error prototype object**.

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

20.5.3 Properties of the Error Prototype Object

The *Error prototype object*:

- is %Error.prototype%.
- is an **ordinary object**.
- is not an Error instance and does not have an [[ErrorData]] internal slot.
- has a [[Prototype]] internal slot whose value is %Object.prototype%.

20.5.3.1 Error.prototype.constructor

The initial value of **Error.prototype.constructor** is %Error%.

20.5.3.2 Error.prototype.message

The initial value of **Error.prototype.message** is the empty String.

20.5.3.3 Error.prototype.name

The initial value of **Error.prototype.name** is "Error".

20.5.3.4 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 `? Get(O, "name")`.
4. If *name* is **undefined**, set *name* to "Error"; otherwise set *name* to `? ToString(name)`.
5. Let *msg* be `? Get(O, "message")`.
6. If *msg* is **undefined**, set *msg* to the empty String; otherwise set *msg* to `? ToString(msg)`.
7. If *name* is the empty String, return *msg*.
8. If *msg* is the empty String, return *name*.
9. Return the **string-concatenation** of *name*, the code unit 0x003A (COLON), the code unit 0x0020 (SPACE), and *msg*.

20.5.4 Properties of Error Instances

Error instances are ordinary objects that inherit properties from the **Error prototype object** and have an `[[ErrorData]]` internal slot whose value is **undefined**. The only specified uses of `[[ErrorData]]` is to identify Error, AggregateError, and *NativeError* instances as Error objects within **Object.prototype.toString**.

20.5.5 Native Error Types Used in This Standard

A new instance of one of the *NativeError* objects below or of the AggregateError object is thrown when a runtime error is detected. All *NativeError* objects share the same structure, as described in 20.5.6.

20.5.5.1 EvalError

The EvalError **constructor** is `%EvalError%`.

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

20.5.5.2 RangeError

The RangeError **constructor** is `%RangeError%`.

Indicates a value that is not in the set or range of allowable values.

20.5.5.3 ReferenceError

The ReferenceError **constructor** is `%ReferenceError%`.

Indicate that an invalid reference has been detected.

20.5.5.4 SyntaxError

The SyntaxError **constructor** is `%SyntaxError%`.

Indicates that a parsing error has occurred.

20.5.5.5 TypeError

The `TypeError` constructor is `%TypeError%`.

`TypeError` is used to indicate an unsuccessful operation when none of the other *NativeError* objects are an appropriate indication of the failure cause.

20.5.5.6 `URIError`

The `URIError` constructor is `%URIError%`.

Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition.

20.5.6 *NativeError* Object Structure

When an ECMAScript implementation detects a runtime error, it throws a new instance of one of the *NativeError* objects defined in 20.5.5. 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 20.5.5.

20.5.6.1 The *NativeError* Constructors

Each *NativeError* constructor:

- creates and initializes a new *NativeError* object when called as a function rather than as a `constructor`. A call of the object as a function is equivalent to calling it as a `constructor` with the same arguments. Thus the function call `NativeError(...)` is equivalent to the object creation expression `new NativeError(...)` with the same arguments.
- is designed to be subclassable. It may be used as the value of an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified *NativeError* behaviour must include a `super` call to the *NativeError* constructor to create and initialize subclass instances with an `[[ErrorData]]` internal slot.

20.5.6.1.1 *NativeError* (*message*)

When a *NativeError* function is called with argument *message*, the following steps are taken:

1. If `NewTarget` is `undefined`, let *newTarget* be the `active function object`; else let *newTarget* be `NewTarget`.
2. Let *O* be `? OrdinaryCreateFromConstructor(newTarget, "%NativeError.prototype%", « [[ErrorData]] »)`.
3. 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. Perform `! DefinePropertyOrThrow(O, "message", msgDesc)`.
4. Return *O*.

The actual value of the string passed in step 2 is either `"%EvalError.prototype%"`, `"%RangeError.prototype%"`, `"%ReferenceError.prototype%"`, `"%SyntaxError.prototype%"`, `"%TypeError.prototype%"`, or `"%URIError.prototype%"` corresponding to which *NativeError* constructor is being defined.

20.5.6.2 Properties of the *NativeError* Constructors

Each *NativeError* constructor:

- has a `[[Prototype]]` internal slot whose value is `%Error%`.
- has a `"name"` property whose value is the String value `"NativeError"`.
- has the following properties:

20.5.6.2.1 *NativeError*.prototype

The initial value of ***NativeError*.prototype** is a *NativeError* prototype object (20.5.6.3). Each *NativeError* constructor has a distinct prototype object.

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

20.5.6.3 Properties of the *NativeError* Prototype Objects

Each *NativeError* prototype object:

- is an ordinary object.
- is not an Error instance and does not have an `[[ErrorData]]` internal slot.
- has a `[[Prototype]]` internal slot whose value is `%Error.prototype%`.

20.5.6.3.1 *NativeError*.prototype.constructor

The initial value of the `"constructor"` property of the prototype for a given *NativeError* constructor is the corresponding intrinsic object `%NativeError%` (20.5.6.1).

20.5.6.3.2 *NativeError*.prototype.message

The initial value of the `"message"` property of the prototype for a given *NativeError* constructor is the empty String.

20.5.6.3.3 *NativeError*.prototype.name

The initial value of the `"name"` property of the prototype for a given *NativeError* constructor is the String value consisting of the name of the constructor (the name used instead of *NativeError*).

20.5.6.4 Properties of *NativeError* Instances

NativeError instances are ordinary objects that inherit properties from their *NativeError* prototype object and have an `[[ErrorData]]` internal slot whose value is `undefined`. The only specified use of `[[ErrorData]]` is by **Object.prototype.toString** (20.1.3.6) to identify Error, AggregateError, or *NativeError* instances.

20.5.7 AggregateError Objects

20.5.7.1 The AggregateError Constructor

The AggregateError constructor:

- is `%AggregateError%`.
- is the initial value of the `"AggregateError"` property of the global object.

- creates and initializes a new `AggregateError` object when called as a function rather than as a `constructor`. Thus the function call `AggregateError(...)` is equivalent to the object creation expression `new AggregateError(...)` with the same arguments.
- is designed to be subclassable. It may be used as the value of an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `AggregateError` behaviour must include a `super` call to the `AggregateError` `constructor` to create and initialize subclass instances with an `[[ErrorData]]` internal slot.

20.5.7.1.1 `AggregateError (errors, message)`

When the `AggregateError` function is called with arguments `errors` and `message`, the following steps are taken:

1. If `NewTarget` is `undefined`, let `newTarget` be the `active function object`; else let `newTarget` be `NewTarget`.
2. Let `O` be ? `OrdinaryCreateFromConstructor(newTarget, "%AggregateError.prototype%", « [[ErrorData]] »)`.
3. 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. Perform ! `DefinePropertyOrThrow(O, "message", msgDesc)`.
4. Let `errorsList` be ? `IterableToList(errors)`.
5. Perform ! `DefinePropertyOrThrow(O, "errors", PropertyDescriptor { [[Configurable]]: true, [[Enumerable]]: false, [[Writable]]: true, [[Value]]: ! CreateArrayFromList(errorsList) })`.
6. Return `O`.

20.5.7.2 Properties of the `AggregateError` Constructor

The `AggregateError` `constructor`:

- has a `[[Prototype]]` internal slot whose value is `%Error%`.
- has the following properties:

20.5.7.2.1 `AggregateError.prototype`

The initial value of `AggregateError.prototype` is `%AggregateError.prototype%`.

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

20.5.7.3 Properties of the `AggregateError` Prototype Object

The `AggregateError` `prototype` object:

- is `%AggregateError.prototype%`.
- is an `ordinary object`.
- is not an `Error` instance or an `AggregateError` instance and does not have an `[[ErrorData]]` internal slot.
- has a `[[Prototype]]` internal slot whose value is `%Error.prototype%`.

20.5.7.3.1 `AggregateError.prototype.constructor`

The initial value of `AggregateError.prototype.constructor` is `%AggregateError%`.

20.5.7.3.2 `AggregateError.prototype.message`

The initial value of **AggregateError.prototype.message** is the empty String.

20.5.7.3.3 AggregateError.prototype.name

The initial value of **AggregateError.prototype.name** is "AggregateError".

20.5.7.4 Properties of AggregateError Instances

AggregateError instances are ordinary objects that inherit properties from their [AggregateError prototype object](#) and have an `[[ErrorData]]` internal slot whose value is **undefined**. The only specified use of `[[ErrorData]]` is by **Object.prototype.toString** (20.1.3.6) to identify Error, AggregateError, or *NativeError* instances.

21 Numbers and Dates

21.1 Number Objects

21.1.1 The Number Constructor

The Number [constructor](#):

- is *%Number%*.
- is the initial value of the "Number" property of the [global object](#).
- creates and initializes a new Number object when called as a [constructor](#).
- performs a type conversion when called as a function rather than as a [constructor](#).
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified Number behaviour must include a **super** call to the Number [constructor](#) to create and initialize the subclass instance with a `[[NumberData]]` internal slot.

21.1.1.1 Number (*value*)

When **Number** is called with argument *value*, the following steps are taken:

1. If *value* is present, then
 - a. Let *prim* be ? [ToNumeric](#)(*value*).
 - b. If [Type](#)(*prim*) is BigInt, let *n* be $\mathbb{F}(\mathbb{R}(\text{prim}))$.
 - c. Otherwise, let *n* be *prim*.
2. Else,
 - a. Let *n* be $+0_{\mathbb{F}}$.
3. If `NewTarget` is **undefined**, return *n*.
4. Let *O* be ? [OrdinaryCreateFromConstructor](#)(`NewTarget`, `"%Number.prototype%"`, « `[[NumberData]]` »).
5. Set *O*.`[[NumberData]]` to *n*.
6. Return *O*.

21.1.2 Properties of the Number Constructor

The Number [constructor](#):

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

21.1.2.1 `Number.EPSILON`

The value of `Number.EPSILON` is the [Number value](#) for the magnitude of the difference between 1 and the smallest value greater than 1 that is representable as a [Number value](#), which is approximately $2.2204460492503130808472633361816 \times 10^{-16}$.

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

21.1.2.2 `Number.isFinite (number)`

When `Number.isFinite` is called with one argument *number*, the following steps are taken:

1. If `Type(number)` is not `Number`, return `false`.
2. If *number* is `NaN`, $+\infty_{\mathbb{F}}$, or $-\infty_{\mathbb{F}}$, return `false`.
3. Otherwise, return `true`.

21.1.2.3 `Number.isInteger (number)`

When `Number.isInteger` is called with one argument *number*, the following steps are taken:

1. Return `!IsIntegralNumber(number)`.

21.1.2.4 `Number.isNaN (number)`

When `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 ([19.2.3](#)) in that it does not convert its argument to a `Number` before determining whether it is `NaN`.

21.1.2.5 `Number.isSafeInteger (number)`

When `Number.isSafeInteger` is called with one argument *number*, the following steps are taken:

1. If `!IsIntegralNumber(number)` is `true`, then
 - a. If $\text{abs}(\mathbb{R}(number)) \leq 2^{53} - 1$, return `true`.
2. Return `false`.

21.1.2.6 `Number.MAX_SAFE_INTEGER`

NOTE The value of **Number.MAX_SAFE_INTEGER** is the largest *integral Number* n such that $\mathbb{R}(n)$ and $\mathbb{R}(n) + 1$ are both exactly representable as a *Number value*.

The value of **Number.MAX_SAFE_INTEGER** is $9007199254740991_{\mathbb{F}}$ ($\mathbb{F}(2^{53} - 1)$).

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

21.1.2.7 Number.MAX_VALUE

The value of **Number.MAX_VALUE** is the largest positive finite value of the Number type, which is approximately $1.7976931348623157 \times 10^{308}$.

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

21.1.2.8 Number.MIN_SAFE_INTEGER

NOTE The value of **Number.MIN_SAFE_INTEGER** is the smallest *integral Number* n such that $\mathbb{R}(n)$ and $\mathbb{R}(n) - 1$ are both exactly representable as a *Number value*.

The value of **Number.MIN_SAFE_INTEGER** is $-9007199254740991_{\mathbb{F}}$ ($\mathbb{F}(-(2^{53} - 1))$).

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

21.1.2.9 Number.MIN_VALUE

The value of **Number.MIN_VALUE** is the smallest positive value of the Number type, which is approximately 5×10^{-324} .

In the [IEEE 754-2019](#) 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** }.

21.1.2.10 Number.NaN

The value of **Number.NaN** is NaN.

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

21.1.2.11 Number.NEGATIVE_INFINITY

The value of **Number.NEGATIVE_INFINITY** is $-\infty_{\mathbb{F}}$.

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

21.1.2.12 Number.parseFloat (*string*)

The value of the **Number.parseFloat** data property is the same built-in function object that is the initial value of the "parseFloat" property of the global object defined in 19.2.4.

21.1.2.13 Number.parseInt (*string*, *radix*)

The value of the **Number.parseInt** data property is the same built-in function object that is the initial value of the "parseInt" property of the global object defined in 19.2.5.

21.1.2.14 Number.POSITIVE_INFINITY

The value of **Number.POSITIVE_INFINITY** is $+\infty_{\mathbb{F}}$.

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

21.1.2.15 Number.prototype

The initial value of **Number.prototype** is the Number prototype object.

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

21.1.3 Properties of the Number Prototype Object

The *Number prototype object*:

- is *%Number.prototype%*.
- is an ordinary object.
- is itself a Number object; it has a [[NumberData]] internal slot with the value $+0_{\mathbb{F}}$.
- has a [[Prototype]] internal slot whose value is *%Object.prototype%*.

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* takes argument *value*. It performs the following steps when called:

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 *value*.[[NumberData]].
 - b. **Assert**: **Type**(*n*) is Number.
 - c. 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.

21.1.3.1 Number.prototype.constructor

The initial value of **Number.prototype.constructor** is *%Number%*.

21.1.3.2 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. Let f be ? [ToIntegerOrInfinity](#)(*fractionDigits*).
3. **Assert**: If *fractionDigits* is **undefined**, then f is 0.
4. If x is not finite, return ! [Number::toString](#)(x).
5. If $f < 0$ or $f > 100$, throw a **RangeError** exception.
6. Set x to $\mathbb{R}(x)$.
7. Let s be the empty String.
8. If $x < 0$, then
 - a. Set s to "-".
 - b. Set x to $-x$.
9. If $x = 0$, then
 - a. Let m be the String value consisting of $f + 1$ occurrences of the code unit 0x0030 (DIGIT ZERO).
 - b. Let e be 0.
10. Else,
 - 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 $n \times 10^{e-n} - x$ is as close to zero as possible. If there are two such sets of e and n , pick the e and n for which $n \times 10^{e-f}$ is larger.
 - b. Else,
 - i. Let e , n , and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, $n \times 10^{e-f}$ is x , and f is as small as possible. Note that the decimal representation of n has $f + 1$ digits, n is not divisible by 10, and the least significant digit of n is not necessarily uniquely determined by these criteria.
 - c. Let m be the String value consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
11. If $f \neq 0$, then
 - a. Let a be the first code unit of m .
 - b. Let b be the other f code units of m .
 - c. Set m to the [string-concatenation](#) of a , ".", and b .
12. If $e = 0$, then
 - a. Let c be "+".
 - b. Let d be "0".
13. Else,
 - a. If $e > 0$, let c be "+".
 - b. Else,
 - i. **Assert**: $e < 0$.
 - ii. Let c be "-".
 - iii. Set e to $-e$.
 - c. Let d be the String value consisting of the digits of the decimal representation of e (in order, with no leading zeroes).
14. Set m to the [string-concatenation](#) of m , "e", c , and d .
15. Return the [string-concatenation](#) of s and m .

NOTE

For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 10.b.i be used as a guideline:

- i. Let e , n , and f be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, $n \times 10^{e-f}$ is x , and f is as small as possible. If there are multiple possibilities for n , choose the value of n for which $n \times 10^{e-f}$ is closest in value to x . If there are two such possible values of n , choose the one that is even.

21.1.3.3 Number.prototype.toFixed (*fractionDigits*)

NOTE 1

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. Let f be ? **ToIntegerOrInfinity**(*fractionDigits*).
3. **Assert**: If *fractionDigits* is **undefined**, then f is 0.
4. If f is not finite, throw a **RangeError** exception.
5. If $f < 0$ or $f > 100$, throw a **RangeError** exception.
6. If x is not finite, return ! **Number::toString**(x).
7. Set x to $\mathbb{R}(x)$.
8. Let s be the empty String.
9. If $x < 0$, then
 - a. Set s to "-".
 - b. Set x to $-x$.
10. If $x \geq 10^{21}$, then
 - a. Let m be ! **ToString**($\mathbb{F}(x)$).
11. Else,
 - a. Let n be an integer for which $n / 10^f - x$ 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 value consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
 - c. If $f \neq 0$, then
 - i. Let k be the length of m .
 - ii. If $k \leq f$, then
 1. Let z be the String value consisting of $f + 1 - k$ occurrences of the code unit 0x0030 (DIGIT ZERO).
 2. Set m to the string-concatenation of z and m .
 3. Set k to $f + 1$.
 - iii. Let a be the first $k - f$ code units of m .
 - iv. Let b be the other f code units of m .
 - v. Set m to the string-concatenation of a , ".", and b .
12. Return the string-concatenation of s and m .

NOTE 2

The output of **toFixed** may be more precise than **toString** for some values because **toString** only prints enough significant digits to distinguish the number from adjacent Number values. For example,

(1000000000000000128).toString() returns "1000000000000000100", while
(1000000000000000128).toFixed(0) returns "1000000000000000128".

21.1.3.4 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.

Produces a String value that represents this **Number value** formatted according to the conventions of the **host environment's** current locale. This function is **implementation-defined**, and it is permissible, but not encouraged, for it to return the same thing as **toString**.

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.

21.1.3.5 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** instead. Specifically, perform the following steps:

1. Let *x* be ? **thisNumberValue**(**this** value).
2. If *precision* is **undefined**, return ! **ToString**(*x*).
3. Let *p* be ? **ToIntegerOrInfinity**(*precision*).
4. If *x* is not finite, return ! **Number::toString**(*x*).
5. If *p* < 1 or *p* > 100, throw a **RangeError** exception.
6. Set *x* to **ℝ**(*x*).
7. Let *s* be the empty String.
8. If *x* < 0, then
 - a. Set *s* to the code unit 0x002D (HYPHEN-MINUS).
 - b. Set *x* to -*x*.
9. If *x* = 0, then
 - a. Let *m* be the String value consisting of *p* occurrences of the code unit 0x0030 (DIGIT ZERO).
 - b. Let *e* be 0.
10. Else,
 - a. Let *e* and *n* be integers such that $10^{p-1} \leq n < 10^p$ and for which $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-p+1}$ is larger.
 - b. Let *m* be the String value 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. If $p \neq 1$, then
 - 1. Let a be the first code unit of m .
 - 2. Let b be the other $p - 1$ code units of m .
 - 3. Set m to the [string-concatenation](#) of a , ".", and b .
 - iii. If $e > 0$, then
 - 1. Let c be the code unit 0x002B (PLUS SIGN).
 - iv. Else,
 - 1. [Assert](#): $e < 0$.
 - 2. Let c be the code unit 0x002D (HYPHEN-MINUS).
 - 3. Set e to $-e$.
 - v. Let d be the String value consisting of the digits of the decimal representation of e (in order, with no leading zeroes).
 - vi. Return the [string-concatenation](#) of s , m , the code unit 0x0065 (LATIN SMALL LETTER E), c , and d .
11. If $e = p - 1$, return the [string-concatenation](#) of s and m .
 12. If $e \geq 0$, then
 - a. Set m to the [string-concatenation](#) of the first $e + 1$ code units of m , the code unit 0x002E (FULL STOP), and the remaining $p - (e + 1)$ code units of m .
 13. Else,
 - a. Set m to the [string-concatenation](#) of the code unit 0x0030 (DIGIT ZERO), the code unit 0x002E (FULL STOP), $-(e + 1)$ occurrences of the code unit 0x0030 (DIGIT ZERO), and the String m .
 14. Return the [string-concatenation](#) of s and m .

21.1.3.6 Number.prototype.toString ([*radix*])

NOTE The optional *radix* should be an [integral Number](#) value in the inclusive range 2_{F} to 36_{F} . If *radix* is **undefined** then 10_{F} is used as the value of *radix*.

The following steps are performed:

1. Let x be ? [thisNumberValue](#)(**this** value).
2. If *radix* is **undefined**, let *radixMV* be 10.
3. Else, let *radixMV* be ? [ToIntegerOrInfinity](#)(*radix*).
4. If *radixMV* < 2 or *radixMV* > 36 , throw a **RangeError** exception.
5. If *radixMV* = 10, return ! [ToString](#)(x).
6. Return the String representation of this [Number value](#) using the radix specified by *radixMV*. Letters **a-z** are used for digits with values 10 through 35. The precise algorithm is [implementation-defined](#), however the algorithm should be a generalization of that specified in [6.1.6.1.20](#).

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.

The "length" property of the **toString** method is 1_{F} .

21.1.3.7 Number.prototype.valueOf ()

1. Return ? [thisNumberValue](#)(**this** value).

21.1.4 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.

21.2 BigInt Objects

21.2.1 The BigInt Constructor

The `BigInt` [constructor](#):

- is `%BigInt%`.
- is the initial value of the `"BigInt"` property of the [global object](#).
- performs a type conversion when called as a function rather than as a [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 definition but a `super` call to the `BigInt` [constructor](#) will cause an exception.

21.2.1.1 `BigInt (value)`

When `BigInt` is called with argument `value`, the following steps are taken:

1. If `NewTarget` is not `undefined`, throw a `TypeError` exception.
2. Let `prim` be `? ToPrimitive(value, number)`.
3. If `Type(prim)` is `Number`, return `? NumberToBigInt(prim)`.
4. Otherwise, return `? ToBigInt(value)`.

21.2.1.1.1 `NumberToBigInt (number)`

The abstract operation `NumberToBigInt` takes argument `number` (a `Number`). It performs the following steps when called:

1. If `IsIntegralNumber(number)` is `false`, throw a `RangeError` exception.
2. Return the `BigInt` value that represents $\mathbb{R}(number)$.

21.2.2 Properties of the BigInt Constructor

The value of the `[[Prototype]]` internal slot of the `BigInt` [constructor](#) is `%Function.prototype%`.

The `BigInt` [constructor](#) has the following properties:

21.2.2.1 `BigInt.asIntN (bits, bigint)`

When the `BigInt.asIntN` function is called with two arguments `bits` and `bigint`, the following steps are taken:

1. Set `bits` to `? ToIndex(bits)`.
2. Set `bigint` to `? ToBigInt(bigint)`.
3. Let `mod` be $\mathbb{R}(bigint)$ modulo 2^{bits} .

4. If $mod \geq 2^{bits-1}$, return $\mathbb{Z}(mod - 2^{bits})$; otherwise, return $\mathbb{Z}(mod)$.

21.2.2.2 BigInt.asUintN (bits, bigint)

When the **BigInt.asUintN** function is called with two arguments *bits* and *bigint*, the following steps are taken:

1. Set *bits* to ? **ToIndex**(*bits*).
2. Set *bigint* to ? **ToBigInt**(*bigint*).
3. Return the BigInt value that represents $\mathbb{R}(bigint) \bmod 2^{bits}$.

21.2.2.3 BigInt.prototype

The initial value of **BigInt.prototype** is the **BigInt prototype object**.

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

21.2.3 Properties of the BigInt Prototype Object

The *BigInt prototype object*:

- is %*BigInt.prototype*%.
- is an **ordinary object**.
- is not a BigInt object; it does not have a **[[BigIntData]]** internal slot.
- has a **[[Prototype]]** internal slot whose value is %*Object.prototype*%.

The abstract operation *thisBigIntValue* takes argument *value*. It performs the following steps when called:

1. If **Type**(*value*) is BigInt, return *value*.
2. If **Type**(*value*) is Object and *value* has a **[[BigIntData]]** internal slot, then
 - a. **Assert**: **Type**(*value*.**[[BigIntData]]**) is BigInt.
 - b. Return *value*.**[[BigIntData]]**.
3. Throw a **TypeError** exception.

The phrase “this BigInt value” within the specification of a method refers to the result returned by calling the abstract operation **thisBigIntValue** with the **this** value of the method invocation passed as the argument.

21.2.3.1 BigInt.prototype.constructor

The initial value of **BigInt.prototype.constructor** is %*BigInt*%.

21.2.3.2 BigInt.prototype.toLocaleString ([reserved1 [, reserved2]])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the **BigInt.prototype.toLocaleString** method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the **toLocaleString** method is used.

Produces a String value that represents this BigInt value formatted according to the conventions of the **host environment**'s current locale. This function is **implementation-defined**, and it is permissible, but not encouraged, for it to return the same thing as **toString**.

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.

21.2.3.3 `BigInt.prototype.toString ([radix])`

NOTE The optional *radix* should be an [integral Number](#) value in the inclusive range 2_{F} to 36_{F} . If *radix* is **undefined** then 10_{F} is used as the value of *radix*.

The following steps are performed:

1. Let *x* be ? [thisBigIntValue](#)(**this** value).
2. If *radix* is **undefined**, let *radixMV* be 10.
3. Else, let *radixMV* be ? [ToIntegerOrInfinity](#)(*radix*).
4. If *radixMV* < 2 or *radixMV* > 36, throw a **RangeError** exception.
5. If *radixMV* = 10, return ! [ToString](#)(*x*).
6. Return the String representation of this [Number value](#) using the radix specified by *radixMV*. Letters **a-z** are used for digits with values 10 through 35. The precise algorithm is [implementation-defined](#), however the algorithm should be a generalization of that specified in [6.1.6.2.23](#).

The **toString** function is not generic; it throws a **TypeError** exception if its **this** value is not a `BigInt` or a `BigInt` object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

21.2.3.4 `BigInt.prototype.valueOf ()`

1. Return ? [thisBigIntValue](#)(**this** value).

21.2.3.5 `BigInt.prototype [@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value **"BigInt"**.

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

21.3 The Math Object

The Math object:

- is `%Math%`.
- is the initial value of the **"Math"** property of the [global object](#).
- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is not a [function object](#).
- does not have a `[[Construct]]` internal method; it cannot be used as a [constructor](#) with the **new** operator.
- does not have a `[[Call]]` internal method; it cannot be invoked as a function.

NOTE In this specification, the phrase “the [Number value](#) for *x*” has a technical meaning defined in [6.1.6.1](#).

21.3.1 Value Properties of the Math Object

21.3.1.1 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` }.

21.3.1.2 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` }.

21.3.1.3 Math.LN2

The **Number value** for the natural logarithm of 2, which is approximately 0.6931471805599453.

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

21.3.1.4 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**.

21.3.1.5 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**.

21.3.1.6 Math.PI

The **Number value** for π , the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

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

21.3.1.7 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**.

21.3.1.8 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` }.

21.3.1.9 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` }.

21.3.2 Function Properties of the Math Object

NOTE The behaviour of the functions **acos**, **acosh**, **asin**, **asinh**, **atan**, **atanh**, **atan2**, **cbrt**, **cos**, **cosh**, **exp**, **expm1**, **hypot**, **log**, **log1p**, **log2**, **log10**, **pow**, **random**, **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 754-2019](#) arithmetic contained in **fdlibm**, the freely distributable mathematical library from Sun Microsystems (<http://www.netlib.org/fdlibm>).

21.3.2.1 Math.abs (*x*)

Returns the absolute value of *x*; the result has the same magnitude as *x* but has positive sign.

When the **Math.abs** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? `ToNumber(x)`.
2. If *n* is **NaN**, return **NaN**.
3. If *n* is $-0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If *n* is $-\infty_{\mathbb{F}}$, return $+\infty_{\mathbb{F}}$.
5. If *n* < $+0_{\mathbb{F}}$, return $-n$.
6. Return *n*.

21.3.2.2 Math.acos (*x*)

Returns the inverse cosine of x . The result is expressed in radians and ranges from $+0_{\mathbb{F}}$ to $\mathbb{F}(\pi)$, inclusive.

When the **Math.acos** method is called with argument x , the following steps are taken:

1. Let n be ? `ToNumber`(x).
2. If n is **NaN**, $n > 1_{\mathbb{F}}$, or $n < -1_{\mathbb{F}}$, return **NaN**.
3. If n is $1_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. Return an **implementation-approximated** value representing the result of the inverse cosine of $\mathbb{R}(n)$.

21.3.2.3 Math.acosh (x)

Returns the inverse hyperbolic cosine of x .

When the **Math.acosh** method is called with argument x , the following steps are taken:

1. Let n be ? `ToNumber`(x).
2. If n is **NaN** or n is $+\infty_{\mathbb{F}}$, return n .
3. If n is $1_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If $n < 1_{\mathbb{F}}$, return **NaN**.
5. Return an **implementation-approximated** value representing the result of the inverse hyperbolic cosine of $\mathbb{R}(n)$.

21.3.2.4 Math.asin (x)

Returns the inverse sine of x . The result is expressed in radians and ranges from $\mathbb{F}(-\pi / 2)$ to $\mathbb{F}(\pi / 2)$, inclusive.

When the **Math.asin** method is called with argument x , the following steps are taken:

1. Let n be ? `ToNumber`(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$, or n is $-0_{\mathbb{F}}$, return n .
3. If $n > 1_{\mathbb{F}}$ or $n < -1_{\mathbb{F}}$, return **NaN**.
4. Return an **implementation-approximated** value representing the result of the inverse sine of $\mathbb{R}(n)$.

21.3.2.5 Math.asinh (x)

Returns the inverse hyperbolic sine of x .

When the **Math.asinh** method is called with argument x , the following steps are taken:

1. Let n be ? `ToNumber`(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$, n is $-0_{\mathbb{F}}$, n is $+\infty_{\mathbb{F}}$, or n is $-\infty_{\mathbb{F}}$, return n .
3. Return an **implementation-approximated** value representing the result of the inverse hyperbolic sine of $\mathbb{R}(n)$.

21.3.2.6 Math.atan (x)

Returns the inverse tangent of x . The result is expressed in radians and ranges from $\mathbb{F}(-\pi / 2)$ to $\mathbb{F}(\pi / 2)$, inclusive.

When the **Math.atan** method is called with argument x , the following steps are taken:

1. Let n be ? `ToNumber`(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$, or n is $-0_{\mathbb{F}}$, return n .

3. If n is $+\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $\pi / 2$.
4. If n is $-\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $-\pi / 2$.
5. Return an **implementation-approximated** value representing the result of the inverse tangent of $\mathbb{R}(n)$.

21.3.2.7 Math.atanh (x)

Returns the inverse hyperbolic tangent of x .

When the **Math.atanh** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$ or n is $-0_{\mathbb{F}}$, return n .
3. If $n > 1_{\mathbb{F}}$ or $n < -1_{\mathbb{F}}$, return **NaN**.
4. If n is $1_{\mathbb{F}}$, return $+\infty_{\mathbb{F}}$.
5. If n is $-1_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$.
6. Return an **implementation-approximated** value representing the result of the inverse hyperbolic tangent of $\mathbb{R}(n)$.

21.3.2.8 Math.atan2 (y, x)

Returns the inverse 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 inverse 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 $+\pi$, inclusive.

When the **Math.atan2** method is called with arguments y and x , the following steps are taken:

1. Let ny be ? **ToNumber**(y).
2. Let nx be ? **ToNumber**(x).
3. If ny is **NaN** or nx is **NaN**, return **NaN**.
4. If ny is $+\infty_{\mathbb{F}}$, then
 - a. If nx is $+\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $\pi / 4$.
 - b. If nx is $-\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $3\pi / 4$.
 - c. Return an **implementation-approximated** value representing $\pi / 2$.
5. If ny is $-\infty_{\mathbb{F}}$, then
 - a. If nx is $+\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $-\pi / 4$.
 - b. If nx is $-\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $-3\pi / 4$.
 - c. Return an **implementation-approximated** value representing $-\pi / 2$.
6. If ny is $+0_{\mathbb{F}}$, then
 - a. If $nx > +0_{\mathbb{F}}$ or nx is $+0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
 - b. Return an **implementation-approximated** value representing π .
7. If ny is $-0_{\mathbb{F}}$, then
 - a. If $nx > +0_{\mathbb{F}}$ or nx is $+0_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
 - b. Return an **implementation-approximated** value representing $-\pi$.
8. **Assert**: ny is finite and is neither $+0_{\mathbb{F}}$ nor $-0_{\mathbb{F}}$.
9. If $ny > +0_{\mathbb{F}}$, then
 - a. If nx is $+\infty_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.

- b. If nx is $-\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing π .
 - c. If nx is $+0_{\mathbb{F}}$ or nx is $-0_{\mathbb{F}}$, return an **implementation-approximated** value representing $\pi / 2$.
10. If $ny < +0_{\mathbb{F}}$, then
- a. If nx is $+\infty_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
 - b. If nx is $-\infty_{\mathbb{F}}$, return an **implementation-approximated** value representing $-\pi$.
 - c. If nx is $+0_{\mathbb{F}}$ or nx is $-0_{\mathbb{F}}$, return an **implementation-approximated** value representing $-\pi / 2$.
11. **Assert:** nx is finite and is neither $+0_{\mathbb{F}}$ nor $-0_{\mathbb{F}}$.
12. Return an **implementation-approximated** value representing the result of the inverse tangent of the quotient $\mathbb{R}(ny) / \mathbb{R}(nx)$.

21.3.2.9 Math.cbrt (x)

Returns the cube root of x .

When the **Math.cbrt** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$, n is $-0_{\mathbb{F}}$, n is $+\infty_{\mathbb{F}}$, or n is $-\infty_{\mathbb{F}}$, return n .
3. Return an **implementation-approximated** value representing the result of the cube root of $\mathbb{R}(n)$.

21.3.2.10 Math.ceil (x)

Returns the smallest (closest to $-\infty$) **integral Number** value that is not less than x . If x is already an **integral Number**, the result is x .

When the **Math.ceil** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\mathbb{F}}$, n is $-0_{\mathbb{F}}$, n is $+\infty_{\mathbb{F}}$, or n is $-\infty_{\mathbb{F}}$, return n .
3. If $n < +0_{\mathbb{F}}$ and $n > -1_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
4. If n is an **integral Number**, return n .
5. Return the smallest (closest to $-\infty$) **integral Number** value that is not less than n .

NOTE The value of **Math.ceil(x)** is the same as the value of **-Math.floor(-x)**.

21.3.2.11 Math.clz32 (x)

When the **Math.clz32** method is called with argument x , the following steps are taken:

1. Let n be ? **ToUint32**(x).
2. Let p be the number of leading zero bits in the unsigned 32-bit binary representation of n .
3. Return **F**(p).

NOTE If n is $+0_{\mathbb{F}}$ or n is $-0_{\mathbb{F}}$, this method returns $32_{\mathbb{F}}$. If the most significant bit of the 32-bit binary encoding of n is 1, this method returns $+0_{\mathbb{F}}$.

21.3.2.12 **Math.cos** (*x*)

Returns the cosine of *x*. The argument is expressed in radians.

When the **Math.cos** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is **NaN**, *n* is **+0_F** or *n* is **-0_F**, return *n*.
3. If *n* is **+∞_F** or *n* is **-∞_F**, return **NaN**.
4. Return an **implementation-approximated** value representing the result of the cosine of $\mathbb{R}(n)$.

21.3.2.13 **Math.cosh** (*x*)

Returns the hyperbolic cosine of *x*.

When the **Math.cosh** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is **NaN**, *n* is **+∞_F** or *n* is **-∞_F**, return *n*.
3. If *n* is **+0_F** or *n* is **-0_F**, return **1_F**.
4. Return an **implementation-approximated** value representing the result of the hyperbolic cosine of $\mathbb{R}(n)$.

NOTE

The value of **Math.cosh(x)** is the same as the value of **(Math.exp(x) + Math.exp(-x)) / 2**.

21.3.2.14 **Math.exp** (*x*)

Returns the exponential function of *x* (*e* raised to the power of *x*, where *e* is the base of the natural logarithms).

When the **Math.exp** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is **NaN** or *n* is **+∞_F**, return *n*.
3. If *n* is **+0_F** or *n* is **-0_F**, return **1_F**.
4. If *n* is **-∞_F**, return **+0_F**.
5. Return an **implementation-approximated** value representing the result of the exponential function of $\mathbb{R}(n)$.

21.3.2.15 **Math.expm1** (*x*)

Returns the result of 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.

When the **Math.expm1** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is **NaN**, *n* is **+0_F**, *n* is **-0_F** or *n* is **+∞_F**, return *n*.
3. If *n* is **-∞_F**, return **-1_F**.
4. Return an **implementation-approximated** value representing the result of subtracting 1 from the exponential function of $\mathbb{R}(n)$.

21.3.2.16 Math.floor (*x*)

Returns the greatest (closest to $+\infty$) [integral Number](#) value that is not greater than *x*. If *x* is already an [integral Number](#), the result is *x*.

When the **Math.floor** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? [ToNumber](#)(*x*).
2. If *n* is **NaN**, *n* is $+0_{\mathbb{F}}$, *n* is $-0_{\mathbb{F}}$, *n* is $+\infty_{\mathbb{F}}$, or *n* is $-\infty_{\mathbb{F}}$, return *n*.
3. If *n* < $1_{\mathbb{F}}$ and *n* > $+0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If *n* is an [integral Number](#), return *n*.
5. Return the greatest (closest to $+\infty$) [integral Number](#) value that is not greater than *n*.

NOTE The value of **Math.floor(x)** is the same as the value of **-Math.ceil(-x)**.

21.3.2.17 Math.fround (*x*)

When the **Math.fround** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? [ToNumber](#)(*x*).
2. If *n* is **NaN**, return **NaN**.
3. If *n* is one of $+0_{\mathbb{F}}$, $-0_{\mathbb{F}}$, $+\infty_{\mathbb{F}}$, or $-\infty_{\mathbb{F}}$, return *n*.
4. Let *n32* be the result of converting *n* to a value in [IEEE 754-2019](#) binary32 format using roundTiesToEven mode.
5. Let *n64* be the result of converting *n32* to a value in [IEEE 754-2019](#) binary64 format.
6. Return the ECMAScript [Number value](#) corresponding to *n64*.

21.3.2.18 Math.hypot (...*args*)

Returns the square root of the sum of squares of its arguments.

When the **Math.hypot** method is called with zero or more arguments which form the rest parameter ...*args*, the following steps are taken:

1. Let *coerced* be a new empty [List](#).
2. For each element *arg* of *args*, do
 - a. Let *n* be ? [ToNumber](#)(*arg*).
 - b. Append *n* to *coerced*.
3. Let *onlyZero* be **true**.
4. For each element *number* of *coerced*, do
 - a. If *number* is **NaN** or *number* is $+\infty_{\mathbb{F}}$, return *number*.
 - b. If *number* is $-\infty_{\mathbb{F}}$, return $+\infty_{\mathbb{F}}$.
 - c. If *number* is neither $+0_{\mathbb{F}}$ nor $-0_{\mathbb{F}}$, set *onlyZero* to **false**.
5. If *onlyZero* is **true**, return $+0_{\mathbb{F}}$.
6. Return an [implementation-approximated](#) value representing the square root of the sum of squares of the mathematical values of the elements of *coerced*.

The "length" property of the **hypot** method is $2_{\mathbb{F}}$.

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 two or more arguments.

21.3.2.19 Math.imul (*x*, *y*)

When **Math.imul** is called with arguments *x* and *y*, the following steps are taken:

1. Let *a* be $\mathbb{R}(? \text{ ToUint32}(x))$.
2. Let *b* be $\mathbb{R}(? \text{ ToUint32}(y))$.
3. Let *product* be $(a \times b)$ modulo 2^{32} .
4. If *product* $\geq 2^{31}$, return $\mathbb{F}(\text{product} - 2^{32})$; otherwise return $\mathbb{F}(\text{product})$.

21.3.2.20 Math.log (*x*)

Returns the natural logarithm of *x*.

When the **Math.log** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is NaN or *n* is $+\infty_{\mathbb{F}}$, return *n*.
3. If *n* is $1_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If *n* is $+0_{\mathbb{F}}$ or *n* is $-0_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$.
5. If *n* < $+0_{\mathbb{F}}$, return NaN.
6. Return an **implementation-approximated** value representing the result of the natural logarithm of $\mathbb{R}(n)$.

21.3.2.21 Math.log1p (*x*)

Returns 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.

When the **Math.log1p** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is NaN, *n* is $+0_{\mathbb{F}}$, *n* is $-0_{\mathbb{F}}$, or *n* is $+\infty_{\mathbb{F}}$, return *n*.
3. If *n* is $-1_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$.
4. If *n* < $-1_{\mathbb{F}}$, return NaN.
5. Return an **implementation-approximated** value representing the result of the natural logarithm of $1 + \mathbb{R}(n)$.

21.3.2.22 Math.log10 (*x*)

Returns the base 10 logarithm of *x*.

When the **Math.log10** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? **ToNumber**(*x*).
2. If *n* is NaN or *n* is $+\infty_{\mathbb{F}}$, return *n*.
3. If *n* is $1_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.

4. If n is $+0_{\mathbb{F}}$ or n is $-0_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$.
5. If $n < +0_{\mathbb{F}}$, return **NaN**.
6. Return an [implementation-approximated](#) value representing the result of the base 10 logarithm of $\mathbb{R}(n)$.

21.3.2.23 **Math.log2 (*x*)**

Returns the base 2 logarithm of x .

When the **Math.log2** method is called with argument x , the following steps are taken:

1. Let n be ? [ToNumber](#)(x).
2. If n is **NaN** or n is $+\infty_{\mathbb{F}}$, return n .
3. If n is $1_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If n is $+0_{\mathbb{F}}$ or n is $-0_{\mathbb{F}}$, return $-\infty_{\mathbb{F}}$.
5. If $n < +0_{\mathbb{F}}$, return **NaN**.
6. Return an [implementation-approximated](#) value representing the result of the base 2 logarithm of $\mathbb{R}(n)$.

21.3.2.24 **Math.max (...*args*)**

Given zero or more arguments, calls [ToNumber](#) on each of the arguments and returns the largest of the resulting values.

When the **Math.max** method is called with zero or more arguments which form the rest parameter *...args*, the following steps are taken:

1. Let *coerced* be a new empty [List](#).
2. For each element *arg* of *args*, do
 - a. Let n be ? [ToNumber](#)(*arg*).
 - b. Append n to *coerced*.
3. Let *highest* be $-\infty_{\mathbb{F}}$.
4. For each element *number* of *coerced*, do
 - a. If *number* is **NaN**, return **NaN**.
 - b. If *number* is $+0_{\mathbb{F}}$ and *highest* is $-0_{\mathbb{F}}$, set *highest* to $+0_{\mathbb{F}}$.
 - c. If *number* > *highest*, set *highest* to *number*.
5. Return *highest*.

NOTE The comparison of values to determine the largest value is done using the [Abstract Relational Comparison](#) algorithm except that $+0_{\mathbb{F}}$ is considered to be larger than $-0_{\mathbb{F}}$.

The "length" property of the **max** method is $2_{\mathbb{F}}$.

21.3.2.25 **Math.min (...*args*)**

Given zero or more arguments, calls [ToNumber](#) on each of the arguments and returns the smallest of the resulting values.

When the **Math.min** method is called with zero or more arguments which form the rest parameter *...args*, the following steps are taken:

1. Let *coerced* be a new empty [List](#).
2. For each element *arg* of *args*, do
 - a. Let *n* be ? [ToNumber](#)(*arg*).
 - b. Append *n* to *coerced*.
3. Let *lowest* be $+\infty_{\mathbb{F}}$.
4. For each element *number* of *coerced*, do
 - a. If *number* is **NaN**, return **NaN**.
 - b. If *number* is $-0_{\mathbb{F}}$ and *lowest* is $+0_{\mathbb{F}}$, set *lowest* to $-0_{\mathbb{F}}$.
 - c. If *number* < *lowest*, set *lowest* to *number*.
5. Return *lowest*.

NOTE The comparison of values to determine the largest value is done using the [Abstract Relational Comparison](#) algorithm except that $+0_{\mathbb{F}}$ is considered to be larger than $-0_{\mathbb{F}}$.

The "length" property of the **min** method is $2_{\mathbb{F}}$.

21.3.2.26 **Math.pow** (*base*, *exponent*)

When the **Math.pow** method is called with arguments *base* and *exponent*, the following steps are taken:

1. Set *base* to ? [ToNumber](#)(*base*).
2. Set *exponent* to ? [ToNumber](#)(*exponent*).
3. Return ! [Number::exponentiate](#)(*base*, *exponent*).

21.3.2.27 **Math.random** ()

Returns a [Number value](#) with positive sign, greater than or equal to $+0_{\mathbb{F}}$ but strictly less than $1_{\mathbb{F}}$, chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an [implementation-defined](#) algorithm or strategy. This function takes no arguments.

Each **Math.random** function created for distinct realms must produce a distinct sequence of values from successive calls.

21.3.2.28 **Math.round** (*x*)

Returns the [Number value](#) that is closest to *x* and is integral. If two integral Numbers are equally close to *x*, then the result is the [Number value](#) that is closer to $+\infty$. If *x* is already integral, the result is *x*.

When the **Math.round** method is called with argument *x*, the following steps are taken:

1. Let *n* be ? [ToNumber](#)(*x*).
2. If *n* is **NaN**, $+\infty_{\mathbb{F}}$, $-\infty_{\mathbb{F}}$ or an [integral Number](#), return *n*.
3. If *n* < $0.5_{\mathbb{F}}$ and *n* > $+0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If *n* < $+0_{\mathbb{F}}$ and *n* ≥ $-0.5_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
5. Return the [integral Number](#) closest to *n*, preferring the Number closer to $+\infty$ in the case of a tie.

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_{F} or is less than $+0_{\text{F}}$ but greater than or equal to -0.5_{F} , **Math.round(x)** returns -0_{F} , but **Math.floor(x + 0.5)** returns $+0_{\text{F}}$. **Math.round(x)** may also differ from the value of **Math.floor(x + 0.5)** because of internal rounding when computing $x + 0.5$.

21.3.2.29 Math.sign (x)

Returns the sign of x , indicating whether x is positive, negative, or zero.

When the **Math.sign** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\text{F}}$, or n is -0_{F} , return n .
3. If $n < +0_{\text{F}}$, return -1_{F} .
4. Return 1_{F} .

21.3.2.30 Math.sin (x)

Returns the sine of x . The argument is expressed in radians.

When the **Math.sin** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\text{F}}$, or n is -0_{F} , return n .
3. If n is $+\infty_{\text{F}}$ or n is $-\infty_{\text{F}}$, return **NaN**.
4. Return an **implementation-approximated** value representing the result of the sine of $\mathbb{R}(n)$.

21.3.2.31 Math.sinh (x)

Returns the hyperbolic sine of x .

When the **Math.sinh** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).
2. If n is **NaN**, n is $+0_{\text{F}}$, n is -0_{F} , n is $+\infty_{\text{F}}$, or n is $-\infty_{\text{F}}$, return n .
3. Return an **implementation-approximated** value representing the result of the hyperbolic sine of $\mathbb{R}(n)$.

NOTE

The value of **Math.sinh(x)** is the same as the value of **(Math.exp(x) - Math.exp(-x)) / 2**.

21.3.2.32 Math.sqrt (x)

Returns the square root of x .

When the **Math.sqrt** method is called with argument x , the following steps are taken:

1. Let n be ? **ToNumber**(x).

2. If n is NaN, n is $+0_{\mathbb{F}}$, n is $-0_{\mathbb{F}}$, or n is $+\infty_{\mathbb{F}}$, return n .
3. If $n < +0_{\mathbb{F}}$, return NaN.
4. Return an [implementation-approximated](#) value representing the result of the square root of $\mathbb{R}(n)$.

21.3.2.33 Math.tan (x)

Returns the tangent of x . The argument is expressed in radians.

When the **Math.tan** method is called with argument x , the following steps are taken:

1. Let n be ? [ToNumber](#)(x).
2. If n is NaN, n is $+0_{\mathbb{F}}$, or n is $-0_{\mathbb{F}}$, return n .
3. If n is $+\infty_{\mathbb{F}}$, or n is $-\infty_{\mathbb{F}}$, return NaN.
4. Return an [implementation-approximated](#) value representing the result of the tangent of $\mathbb{R}(n)$.

21.3.2.34 Math.tanh (x)

Returns the hyperbolic tangent of x .

When the **Math.tanh** method is called with argument x , the following steps are taken:

1. Let n be ? [ToNumber](#)(x).
2. If n is NaN, n is $+0_{\mathbb{F}}$, or n is $-0_{\mathbb{F}}$, return n .
3. If n is $+\infty_{\mathbb{F}}$, return $1_{\mathbb{F}}$.
4. If n is $-\infty_{\mathbb{F}}$, return $-1_{\mathbb{F}}$.
5. Return an [implementation-approximated](#) value representing the result of the hyperbolic tangent of $\mathbb{R}(n)$.

NOTE

The value of **Math.tanh(x)** is the same as the value of **(Math.exp(x) - Math.exp(-x)) / (Math.exp(x) + Math.exp(-x))**.

21.3.2.35 Math.trunc (x)

Returns the integral part of the number x , removing any fractional digits. If x is already integral, the result is x .

When the **Math.trunc** method is called with argument x , the following steps are taken:

1. Let n be ? [ToNumber](#)(x).
2. If n is NaN, n is $+0_{\mathbb{F}}$, n is $-0_{\mathbb{F}}$, n is $+\infty_{\mathbb{F}}$, or n is $-\infty_{\mathbb{F}}$, return n .
3. If $n < 1_{\mathbb{F}}$ and $n > +0_{\mathbb{F}}$, return $+0_{\mathbb{F}}$.
4. If $n < +0_{\mathbb{F}}$ and $n > -1_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
5. Return the [integral Number](#) nearest n in the direction of $+0_{\mathbb{F}}$.

21.4 Date Objects

21.4.1 Overview of Date Objects and Definitions of Abstract Operations

The following functions are [abstract operations](#) that operate on time values (defined in 21.4.1.1). Note that, in every

case, if any argument to one of these functions is **NaN**, the result will be **NaN**.

21.4.1.1 Time Values and Time Range

Time measurement in ECMAScript is analogous to time measurement in POSIX, in particular sharing definition in terms of the proleptic Gregorian calendar, an epoch of midnight at the beginning of 1 January 1970 UTC, and an accounting of every day as comprising exactly 86,400 seconds (each of which is 1000 milliseconds long).

An ECMAScript *time value* is a Number, either a finite **integral Number** representing an instant in time to millisecond precision or **NaN** representing no specific instant. A time value that is a multiple of $24 \times 60 \times 60 \times 1000 = 86,400,000$ (i.e., is equal to $86,400,000 \times d$ for some **integer** d) represents the instant at the start of the UTC day that follows the epoch by d whole UTC days (preceding the epoch for negative d). Every other finite time value t is defined relative to the greatest preceding time value s that is such a multiple, and represents the instant that occurs within the same UTC day as s but follows it by $t - s$ milliseconds.

Time values do not account for UTC leap seconds—there are no time values representing instants within positive leap seconds, and there are time values representing instants removed from the UTC timeline by negative leap seconds. However, the definition of time values nonetheless yields piecewise alignment with UTC, with discontinuities only at leap second boundaries and zero difference outside of leap seconds.

A Number can exactly represent all integers from -9,007,199,254,740,992 to 9,007,199,254,740,992 (21.1.2.8 and 21.1.2.6). A time value supports a slightly smaller range of -8,640,000,000,000,000 to 8,640,000,000,000,000 milliseconds. This yields a supported time value range of exactly -100,000,000 days to 100,000,000 days relative to midnight at the beginning of 1 January 1970 UTC.

The exact moment of midnight at the beginning of 1 January 1970 UTC is represented by the time value **+0_F**.

NOTE The 400 year cycle of the proleptic Gregorian calendar contains 97 leap years. This yields an average of 365.2425 days per year, which is 31,556,952,000 milliseconds. Therefore, the maximum range a Number could represent exactly with millisecond precision is approximately -285,426 to 285,426 years relative to 1970. The smaller range supported by a time value as specified in this section is approximately -273,790 to 273,790 years relative to 1970.

21.4.1.2 Day Number and Time within Day

A given **time value** t belongs to day number

$$\text{Day}(t) = \mathbb{F}(\text{floor}(\mathbb{R}(t / \text{msPerDay})))$$

where the number of milliseconds per day is

$$\text{msPerDay} = \mathbf{86400000}_{\mathbb{F}}$$

The remainder is called the time within the day:

$$\text{TimeWithinDay}(t) = \mathbb{F}(\mathbb{R}(t) \bmod \mathbb{R}(\text{msPerDay}))$$

21.4.1.3 Year Number

ECMAScript uses a proleptic Gregorian calendar to map a day number to a year number and to determine the month

and date within that year. In this calendar, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number y is therefore defined by

$$\begin{aligned}
 \text{DaysInYear}(y) &= 365_{\mathbb{F}} \text{ if } (\mathbb{R}(y) \bmod 4) \neq 0 \\
 &= 366_{\mathbb{F}} \text{ if } (\mathbb{R}(y) \bmod 4) = 0 \text{ and } (\mathbb{R}(y) \bmod 100) \neq 0 \\
 &= 365_{\mathbb{F}} \text{ if } (\mathbb{R}(y) \bmod 100) = 0 \text{ and } (\mathbb{R}(y) \bmod 400) \neq 0 \\
 &= 366_{\mathbb{F}} \text{ if } (\mathbb{R}(y) \bmod 400) = 0
 \end{aligned}$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year y is given by:

$$\text{DayFromYear}(y) = \mathbb{F}(365 \times (\mathbb{R}(y) - 1970) + \text{floor}((\mathbb{R}(y) - 1969) / 4) - \text{floor}((\mathbb{R}(y) - 1901) / 100) + \text{floor}((\mathbb{R}(y) - 1601) / 400))$$

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:

$$\text{YearFromTime}(t) = \text{the largest integral Number } y \text{ (closest to } +\infty) \text{ such that } \text{TimeFromYear}(y) \leq t$$

The leap-year function is $1_{\mathbb{F}}$ for a time within a leap year and otherwise is $+0_{\mathbb{F}}$:

$$\begin{aligned}
 \text{InLeapYear}(t) &= +0_{\mathbb{F}} \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 365_{\mathbb{F}} \\
 &= 1_{\mathbb{F}} \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 366_{\mathbb{F}}
 \end{aligned}$$

21.4.1.4 Month Number

Months are identified by an **integral Number** in the range $+0_{\mathbb{F}}$ to $11_{\mathbb{F}}$ inclusive. The mapping **MonthFromTime**(t) from a **time value** t to a month number is defined by:

$$\begin{aligned}
 \text{MonthFromTime}(t) &= +0_{\mathbb{F}} \text{ if } +0_{\mathbb{F}} \leq \text{DayWithinYear}(t) < 31_{\mathbb{F}} \\
 &= 1_{\mathbb{F}} \text{ if } 31_{\mathbb{F}} \leq \text{DayWithinYear}(t) < 59_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 2_{\mathbb{F}} \text{ if } 59_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 3_{\mathbb{F}} \text{ if } 90_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 4_{\mathbb{F}} \text{ if } 120_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 5_{\mathbb{F}} \text{ if } 151_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 6_{\mathbb{F}} \text{ if } 181_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 7_{\mathbb{F}} \text{ if } 212_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 8_{\mathbb{F}} \text{ if } 243_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 9_{\mathbb{F}} \text{ if } 273_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 10_{\mathbb{F}} \text{ if } 304_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334_{\mathbb{F}} + \text{InLeapYear}(t) \\
 &= 11_{\mathbb{F}} \text{ if } 334_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365_{\mathbb{F}} + \text{InLeapYear}(t)
 \end{aligned}$$

where

$$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))$$

A month value of $+0_{\mathbb{F}}$ specifies January; $1_{\mathbb{F}}$ specifies February; $2_{\mathbb{F}}$ specifies March; $3_{\mathbb{F}}$ specifies April; $4_{\mathbb{F}}$ specifies May; $5_{\mathbb{F}}$ specifies June; $6_{\mathbb{F}}$ specifies July; $7_{\mathbb{F}}$ specifies August; $8_{\mathbb{F}}$ specifies September; $9_{\mathbb{F}}$ specifies October; $10_{\mathbb{F}}$ specifies November; and $11_{\mathbb{F}}$ specifies December. Note that $\text{MonthFromTime}(+0_{\mathbb{F}}) = +0_{\mathbb{F}}$, corresponding to Thursday, 1 January 1970.

21.4.1.5 Date Number

A date number is identified by an **integral Number** in the range $1_{\mathbb{F}}$ through $31_{\mathbb{F}}$, inclusive. The mapping $\text{DateFromTime}(t)$ from a **time value** t to a date number is defined by:

$$\begin{aligned} \text{DateFromTime}(t) &= \text{DayWithinYear}(t) + 1_{\mathbb{F}} \text{ if } \text{MonthFromTime}(t) = +0_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 30_{\mathbb{F}} \text{ if } \text{MonthFromTime}(t) = 1_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 58_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 2_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 89_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 3_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 119_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 4_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 150_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 5_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 180_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 6_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 211_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 7_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 242_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 8_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 272_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 9_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 303_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 10_{\mathbb{F}} \\ &= \text{DayWithinYear}(t) - 333_{\mathbb{F}} - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t) = 11_{\mathbb{F}} \end{aligned}$$

21.4.1.6 Week Day

The weekday for a particular **time value** t is defined as

$$\text{WeekDay}(t) = \mathbb{F}(\mathbb{R}(\text{Day}(t) + 4_{\mathbb{F}}) \text{ modulo } 7)$$

A weekday value of $+0_{\mathbb{F}}$ specifies Sunday; $1_{\mathbb{F}}$ specifies Monday; $2_{\mathbb{F}}$ specifies Tuesday; $3_{\mathbb{F}}$ specifies Wednesday; $4_{\mathbb{F}}$ specifies Thursday; $5_{\mathbb{F}}$ specifies Friday; and $6_{\mathbb{F}}$ specifies Saturday. Note that $\text{WeekDay}(+0_{\mathbb{F}}) = 4_{\mathbb{F}}$, corresponding to Thursday, 1 January 1970.

21.4.1.7 LocalTZA (t , $isUTC$)

$\text{LocalTZA}(t, isUTC)$ is an **implementation-defined** algorithm that returns an **integral Number** representing the local time zone adjustment, or offset, in milliseconds. The local political rules for standard time and daylight saving time in effect at t should be used to determine the result in the way specified in this section.

When $isUTC$ is true, $\text{LocalTZA}(t_{UTC}, \text{true})$ should return the offset of the local time zone from UTC measured in milliseconds at time represented by **time value** t_{UTC} . When the result is added to t_{UTC} , it should yield the corresponding Number t_{local} .

When $isUTC$ is false, $\text{LocalTZA}(t_{local}, \text{false})$ should return the offset of the local time zone from UTC measured in

milliseconds at local time represented by Number t_{local} . When the result is subtracted from t_{local} , it should yield the corresponding **time value** t_{UTC} .

Input t is nominally a **time value** but may be any **Number value**. This can occur when *isUTC* is false and t_{local} represents a **time value** that is already offset outside of the **time value** range at the range boundaries. The algorithm must not limit t_{local} to the **time value** range, so that such inputs are supported.

When t_{local} represents local time repeating multiple times at a negative time zone transition (e.g. when the daylight saving time ends or the time zone offset is decreased due to a time zone rule change) or skipped local time at a positive time zone transitions (e.g. when the daylight saving time starts or the time zone offset is increased due to a time zone rule change), t_{local} must be interpreted using the time zone offset before the transition.

If an implementation does not support a conversion described above or if political rules for time t are not available within the implementation, the result must be $+0_{\text{F}}$.

NOTE It is recommended that implementations use the time zone information of the IANA Time Zone Database <https://www.iana.org/time-zones/>.

1:30 AM on 5 November 2017 in America/New_York is repeated twice (fall backward), but it must be interpreted as 1:30 AM UTC-04 instead of 1:30 AM UTC-05.

`LocalTZA(TimeClip(MakeDate(MakeDay(2017, 10, 5), MakeTime(1, 30, 0, 0))), false)` is $-4 \times \text{msPerHour}$.

2:30 AM on 12 March 2017 in America/New_York does not exist, but it must be interpreted as 2:30 AM UTC-05 (equivalent to 3:30 AM UTC-04).

`LocalTZA(TimeClip(MakeDate(MakeDay(2017, 2, 12), MakeTime(2, 30, 0, 0))), false)` is $-5 \times \text{msPerHour}$.

Local time zone offset values may be positive *or* negative.

21.4.1.8 LocalTime (t)

The abstract operation LocalTime takes argument t . It converts t from UTC to local time. It performs the following steps when called:

1. Return $t + \text{LocalTZA}(t, \text{true})$.

NOTE Two different input time values t_{UTC} are converted to the same local time t_{local} at a negative time zone transition when there are repeated times (e.g. the daylight saving time ends or the time zone adjustment is decreased.).

`LocalTime(UTC(t_{local}))` is not necessarily always equal to t_{local} . Correspondingly, `UTC(LocalTime(t_{UTC}))` is not necessarily always equal to t_{UTC} .

21.4.1.9 UTC (t)

The abstract operation UTC takes argument t . It converts t from local time to UTC. It performs the following steps when called:

1. Return $t - \text{LocalTZA}(t, \text{false})$.

NOTE UTC($\text{LocalTime}(t_{\text{UTC}})$) is not necessarily always equal to t_{UTC} . Correspondingly, $\text{LocalTime}(\text{UTC}(t_{\text{local}}))$ is not necessarily always equal to t_{local} .

21.4.1.10 Hours, Minutes, Second, and Milliseconds

The following abstract operations are useful in decomposing time values:

$$\begin{aligned} \text{HourFromTime}(t) &= \mathbb{F}(\text{floor}(\mathbb{R}(t / \text{msPerHour})) \text{ modulo } \text{HoursPerDay}) \\ \text{MinFromTime}(t) &= \mathbb{F}(\text{floor}(\mathbb{R}(t / \text{msPerMinute})) \text{ modulo } \text{MinutesPerHour}) \\ \text{SecFromTime}(t) &= \mathbb{F}(\text{floor}(\mathbb{R}(t / \text{msPerSecond})) \text{ modulo } \text{SecondsPerMinute}) \\ \text{msFromTime}(t) &= \mathbb{F}(\mathbb{R}(t) \text{ modulo } \text{msPerSecond}) \end{aligned}$$

where

$$\begin{aligned} \text{HoursPerDay} &= 24 \\ \text{MinutesPerHour} &= 60 \\ \text{SecondsPerMinute} &= 60 \\ \text{msPerSecond} &= 1000_{\mathbb{F}} \\ \text{msPerMinute} &= 60000_{\mathbb{F}} = \text{msPerSecond} \times \mathbb{F}(\text{SecondsPerMinute}) \\ \text{msPerHour} &= 3600000_{\mathbb{F}} = \text{msPerMinute} \times \mathbb{F}(\text{MinutesPerHour}) \end{aligned}$$

21.4.1.11 MakeTime (*hour*, *min*, *sec*, *ms*)

The abstract operation MakeTime takes arguments *hour* (a Number), *min* (a Number), *sec* (a Number), and *ms* (a Number). It calculates a number of milliseconds. It performs the following steps when called:

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 $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{hour}))$.
3. Let *m* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{min}))$.
4. Let *s* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{sec}))$.
5. Let *milli* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{ms}))$.
6. Let *t* be $((h * \text{msPerHour} + m * \text{msPerMinute}) + s * \text{msPerSecond}) + \text{milli}$, performing the arithmetic according to IEEE 754-2019 rules (that is, as if using the ECMAScript operators * and +).
7. Return *t*.

21.4.1.12 MakeDay (*year*, *month*, *date*)

The abstract operation MakeDay takes arguments *year* (a Number), *month* (a Number), and *date* (a Number). It calculates a number of days. It performs the following steps when called:

1. If *year* is not finite or *month* is not finite or *date* is not finite, return NaN.
2. Let *y* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{year}))$.
3. Let *m* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{month}))$.
4. Let *dt* be $\mathbb{F}(! \text{ToIntegerOrInfinity}(\text{date}))$.
5. Let *ym* be $y + \mathbb{F}(\text{floor}(\mathbb{R}(m) / 12))$.
6. If *ym* is not finite, return NaN.

7. Let mn be $\mathbb{F}(\mathbb{R}(m))$ modulo 12).
8. Find a finite time value t such that $\text{YearFromTime}(t)$ is ym and $\text{MonthFromTime}(t)$ is mn and $\text{DateFromTime}(t)$ is $1_{\mathbb{F}}$; but if this is not possible (because some argument is out of range), return NaN .
9. Return $\text{Day}(t) + dt - 1_{\mathbb{F}}$.

21.4.1.13 MakeDate (*day*, *time*)

The abstract operation MakeDate takes arguments *day* (a Number) and *time* (a Number). It calculates a number of milliseconds. It performs the following steps when called:

1. If *day* is not finite or *time* is not finite, return NaN .
2. Let tv be $day \times \text{msPerDay} + time$.
3. If tv is not finite, return NaN .
4. Return tv .

21.4.1.14 TimeClip (*time*)

The abstract operation TimeClip takes argument *time* (a Number). It calculates a number of milliseconds. It performs the following steps when called:

1. If *time* is not finite, return NaN .
2. If $\text{abs}(\mathbb{R}(time)) > 8.64 \times 10^{15}$, return NaN .
3. Return $\mathbb{F}(! \text{ToIntegerOrInfinity}(time))$.

21.4.1.15 Date Time String Format

ECMAScript defines a string interchange format for date-times based upon a simplification of the ISO 8601 calendar date extended format. The format is as follows: **YYYY-MM-DDTHH:mm:ss.sssZ**

Where the elements are as follows:

- YYYY** is the year in the proleptic Gregorian calendar as four decimal digits from 0000 to 9999, or as an [expanded year](#) of "+" or "-" followed by six decimal digits.
- "-" (hyphen) appears literally twice in the string.
- MM** is the month of the year as two decimal digits from 01 (January) to 12 (December).
- DD** is the day of the month as two decimal digits 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 UTC offset representation specified as "Z" (for UTC with no offset) or an offset of either "+" or "-" followed by a time expression **HH:mm** (indicating local time ahead of or behind UTC, respectively)

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 UTC offset representation appended:

THH:mm
THH:mm:ss
THH:mm:ss.sss

A string containing out-of-bounds or nonconforming elements 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**. This interpretation of the latter form as "end of a calendar day" is consistent with ISO 8601, even though that specification reserves it for describing time intervals and does not permit it within representations of single points in time.

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, both ISO 8601 and this format specify numeric representations of time zone offsets.

21.4.1.15.1 Expanded Years

Covering the full [time value](#) range of approximately 273,790 years forward or backward from 1 January 1970 (21.4.1.1) requires representing years before 0 or after 9999. ISO 8601 permits expansion of the year representation, but only by mutual agreement of the partners in information interchange. In the simplified ECMAScript format, such an expanded year representation shall have 6 digits and is always prefixed with a + or - sign. The year 0 is considered positive and hence prefixed with a + sign. Strings matching the [Date Time String Format](#) with expanded years representing instants in time outside the range of a [time value](#) are treated as unrecognizable by [Date.parse](#) and cause that function to return NaN without falling back to implementation-specific behaviour or heuristics.

NOTE Examples of date-time values with expanded years:

-271821-04-20T00:00:00Z	271822 B.C.
-000001-01-01T00:00:00Z	2 B.C.
+000000-01-01T00:00:00Z	1 B.C.
+000001-01-01T00:00:00Z	1 A.D.
+001970-01-01T00:00:00Z	1970 A.D.
+002009-12-15T00:00:00Z	2009 A.D.
+275760-09-13T00:00:00Z	275760 A.D.

21.4.2 The Date Constructor

The Date [constructor](#):

- is `%Date%`.
- is the initial value of the **"Date"** property of the [global object](#).
- creates and initializes a new Date object when called as a [constructor](#).
- returns a String representing the current time (UTC) when called as a function rather than as a [constructor](#).
- is a function whose behaviour differs based upon the number and types of its arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified Date behaviour must include a **super** call to the Date [constructor](#) to create and initialize the subclass instance with a `[[DateValue]]` internal slot.
- has a **"length"** property whose value is 7_F.

21.4.2.1 Date (...*values*)

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

1. If `NewTarget` is **undefined**, then
 - a. Let *now* be the [time value](#) (UTC) identifying the current time.
 - b. Return `ToDateString(now)`.
2. Let *numberOfArgs* be the number of elements in *values*.
3. If *numberOfArgs* = 0, then
 - a. Let *dv* be the [time value](#) (UTC) identifying the current time.
4. Else if *numberOfArgs* = 1, then
 - a. Let *value* be *values*[0].
 - b. If `Type(value)` is Object and *value* has a `[[DateValue]]` internal slot, then
 - i. Let *tv* be `!thisTimeValue(value)`.
 - c. Else,
 - i. Let *v* be `?ToPrimitive(value)`.
 - ii. If `Type(v)` is String, then
 1. **Assert**: The next step never returns an [abrupt completion](#) because `Type(v)` is String.
 2. Let *tv* be the result of parsing *v* as a date, in exactly the same manner as for the **parse** method (21.4.3.2).
 - iii. Else,
 1. Let *tv* be `?ToNumber(v)`.
 - d. Let *dv* be `TimeClip(tv)`.
5. Else,
 - a. **Assert**: *numberOfArgs* ≥ 2.
 - b. Let *y* be `?ToNumber(values[0])`.
 - c. Let *m* be `?ToNumber(values[1])`.
 - d. If *numberOfArgs* > 2, let *dt* be `?ToNumber(values[2])`; else let *dt* be 1_F.
 - e. If *numberOfArgs* > 3, let *h* be `?ToNumber(values[3])`; else let *h* be +0_F.
 - f. If *numberOfArgs* > 4, let *min* be `?ToNumber(values[4])`; else let *min* be +0_F.
 - g. If *numberOfArgs* > 5, let *s* be `?ToNumber(values[5])`; else let *s* be +0_F.
 - h. If *numberOfArgs* > 6, let *milli* be `?ToNumber(values[6])`; else let *milli* be +0_F.
 - i. If *y* is NaN, let *yr* be NaN.
 - j. Else,

- i. Let *yi* be ! `ToIntegerOrInfinity(y)`.
- ii. If $0 \leq yi \leq 99$, let *yr* be `1900F + F(yi)`; otherwise, let *yr* be *y*.
- k. Let *finalDate* be `MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli))`.
 - l. Let *dv* be `TimeClip(UTC(finalDate))`.
6. Let *O* be ? `OrdinaryCreateFromConstructor(NewTarget, "%Date.prototype%", « [[DateValue]] »)`.
7. Set *O*.[[DateValue]] to *dv*.
8. Return *O*.

21.4.3 Properties of the Date Constructor

The Date `constructor`:

- has a [[Prototype]] internal slot whose value is `%Function.prototype%`.
- has the following properties:

21.4.3.1 Date.now ()

The `now` function returns the `time value` designating the UTC date and time of the occurrence of the call to `now`.

21.4.3.2 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 String according to the format described in `Date Time String Format (21.4.1.15)`, including expanded years. If the String does not conform to that format the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Strings that are unrecognizable or contain out-of-bounds format element values shall cause `Date.parse` to return `NaN`.

If the String conforms to the `Date Time String Format`, substitute values take the place of absent format elements. When the `MM` or `DD` elements are absent, "01" is used. When the `HH`, `mm`, or `ss` elements are absent, "00" is used. When the `sss` element is absent, "000" is used. When the UTC offset representation is absent, date-only forms are interpreted as a UTC time and date-time forms are interpreted as a local time.

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-defined` when given any String value that does not conform to the

Date Time String Format (21.4.1.15) and that could not be produced in that implementation by the **toString** or **toUTCString** method.

21.4.3.3 Date.prototype

The initial value of **Date.prototype** is the [Date prototype object](#).

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

21.4.3.4 Date.UTC (year [, month [, date [, hours [, minutes [, seconds [, ms]]]]])

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

1. Let *y* be ? [ToNumber](#)(*year*).
2. If *month* is present, let *m* be ? [ToNumber](#)(*month*); else let *m* be +0_F.
3. If *date* is present, let *dt* be ? [ToNumber](#)(*date*); else let *dt* be 1_F.
4. If *hours* is present, let *h* be ? [ToNumber](#)(*hours*); else let *h* be +0_F.
5. If *minutes* is present, let *min* be ? [ToNumber](#)(*minutes*); else let *min* be +0_F.
6. If *seconds* is present, let *s* be ? [ToNumber](#)(*seconds*); else let *s* be +0_F.
7. If *ms* is present, let *milli* be ? [ToNumber](#)(*ms*); else let *milli* be +0_F.
8. If *y* is NaN, let *yr* be NaN.
9. Else,
 - a. Let *yi* be ! [ToIntegerOrInfinity](#)(*y*).
 - b. If $0 \leq yi \leq 99$, let *yr* be 1900_F + [F](#)(*yi*); otherwise, let *yr* be *y*.
10. Return [TimeClip](#)([MakeDate](#)([MakeDay](#)(*yr*, *m*, *dt*), [MakeTime](#)(*h*, *min*, *s*, *milli*))).

The "length" property of the **UTC** function is 7_F.

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.

21.4.4 Properties of the Date Prototype Object

The *Date prototype object*:

- is *%Date.prototype%*.
- is itself an [ordinary object](#).
- is not a Date instance and does not have a **[[DateValue]]** internal slot.
- has a **[[Prototype]]** internal slot whose value is *%Object.prototype%*.

Unless explicitly defined otherwise, the methods of the Date prototype object defined below are not generic and the **this** value passed to them must be an object that has a **[[DateValue]]** internal slot that has been initialized to a [time value](#).

The abstract operation *thisTimeValue* takes argument *value*. It performs the following steps when called:

1. If [Type](#)(*value*) is Object and *value* has a **[[DateValue]]** internal slot, then

- a. Return *value*.[[DateValue]].
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.

21.4.4.1 Date.prototype.constructor

The initial value of **Date.prototype.constructor** is *%Date%*.

21.4.4.2 Date.prototype.getDate ()

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. If *t* is NaN, return NaN.
3. Return **DateFromTime**(**LocalTime**(*t*)).

21.4.4.3 Date.prototype.getDay ()

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. If *t* is NaN, return NaN.
3. Return **WeekDay**(**LocalTime**(*t*)).

21.4.4.4 Date.prototype.getFullYear ()

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. If *t* is NaN, return NaN.
3. Return **YearFromTime**(**LocalTime**(*t*)).

21.4.4.5 Date.prototype.getHours ()

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. If *t* is NaN, return NaN.
3. Return **HourFromTime**(**LocalTime**(*t*)).

21.4.4.6 Date.prototype.getMilliseconds ()

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).

2. If t is NaN, return NaN.
3. Return `msFromTime(LocalTime(t))`.

21.4.4.7 Date.prototype.getMinutes ()

The following steps are performed:

1. Let t be ? `thisTimeValue(this value)`.
2. If t is NaN, return NaN.
3. Return `MinFromTime(LocalTime(t))`.

21.4.4.8 Date.prototype.getMonth ()

The following steps are performed:

1. Let t be ? `thisTimeValue(this value)`.
2. If t is NaN, return NaN.
3. Return `MonthFromTime(LocalTime(t))`.

21.4.4.9 Date.prototype.getSeconds ()

The following steps are performed:

1. Let t be ? `thisTimeValue(this value)`.
2. If t is NaN, return NaN.
3. Return `SecFromTime(LocalTime(t))`.

21.4.4.10 Date.prototype.getTime ()

The following steps are performed:

1. Return ? `thisTimeValue(this value)`.

21.4.4.11 Date.prototype.getTimezoneOffset ()

The following steps are performed:

1. Let t be ? `thisTimeValue(this value)`.
2. If t is NaN, return NaN.
3. Return $(t - \text{LocalTime}(t)) / \text{msPerMinute}$.

21.4.4.12 Date.prototype.getUTCDate ()

The following steps are performed:

1. Let t be ? `thisTimeValue(this value)`.
2. If t is NaN, return NaN.
3. Return `DateFromTime(t)`.

21.4.4.13 Date.prototype.getUTCDay ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [WeekDay](#)(t).

21.4.4.14 Date.prototype.getUTCFullYear ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [YearFromTime](#)(t).

21.4.4.15 Date.prototype.getUTCHours ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [HourFromTime](#)(t).

21.4.4.16 Date.prototype.getUTCMilliseconds ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [msFromTime](#)(t).

21.4.4.17 Date.prototype.getUTCMinutes ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [MinFromTime](#)(t).

21.4.4.18 Date.prototype.getUTCMonth ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).
2. If t is NaN, return NaN.
3. Return [MonthFromTime](#)(t).

21.4.4.19 Date.prototype.getUTCSeconds ()

The following steps are performed:

1. Let t be ? [thisTimeValue](#)(**this** value).

2. If t is NaN, return NaN.
3. Return `SecFromTime(t)`.

21.4.4.20 `Date.prototype.setDate (date)`

The following steps are performed:

1. Let t be `LocalTime(? thisTimeValue(this value))`.
2. Let dt be `? ToNumber(date)`.
3. Let $newDate$ be `MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), dt), TimeWithinDay(t))`.
4. Let u be `TimeClip(UTC($newDate$))`.
5. Set the `[[DateValue]]` internal slot of `this Date object` to u .
6. Return u .

21.4.4.21 `Date.prototype.setFullYear (year [, month [, date]])`

The following steps are performed:

1. Let t be `? thisTimeValue(this value)`.
2. If t is NaN, set t to `+0F`; otherwise, set t to `LocalTime(t)`.
3. Let y be `? ToNumber(year)`.
4. If $month$ is not present, let m be `MonthFromTime(t)`; otherwise, let m be `? ToNumber(month)`.
5. If $date$ is not present, let dt be `DateFromTime(t)`; otherwise, let dt be `? ToNumber(date)`.
6. Let $newDate$ be `MakeDate(MakeDay(y , 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 `setFullYear` method is `3F`.

NOTE If $month$ is not present, this method behaves as if $month$ was present with the value `getMonth()`. If $date$ is not present, it behaves as if $date$ was present with the value `getDate()`.

21.4.4.22 `Date.prototype.setHours (hour [, min [, sec [, ms]]])`

The following steps are performed:

1. Let t be `LocalTime(? thisTimeValue(this value))`.
2. Let h be `? ToNumber(hour)`.
3. If min is not present, let m be `MinFromTime(t)`; otherwise, let m be `? ToNumber(min)`.
4. If sec is not present, let s be `SecFromTime(t)`; otherwise, let s be `? ToNumber(sec)`.
5. If ms is not present, let $milli$ be `msFromTime(t)`; otherwise, let $milli$ be `? ToNumber(ms)`.
6. Let $date$ be `MakeDate(Day(t), MakeTime(h , m , s , $milli$))`.
7. Let u be `TimeClip(UTC($date$))`.
8. Set the `[[DateValue]]` internal slot of `this Date object` to u .
9. Return u .

The "length" property of the **setHours** method is 4_F.

NOTE If *min* is not present, this method behaves as if *min* was present with the value **getMinutes()**. If *sec* is not present, it behaves as if *sec* was present with the value **getSeconds()**. If *ms* is not present, it behaves as if *ms* was present with the value **getMilliseconds()**.

21.4.4.23 Date.prototype.setMilliseconds (*ms*)

The following steps are performed:

1. Let *t* be **LocalTime**(? **thisTimeValue**(**this** value)).
2. Set *ms* to ? **ToNumber**(*ms*).
3. Let *time* be **MakeTime**(**HourFromTime**(*t*), **MinFromTime**(*t*), **SecFromTime**(*t*), *ms*).
4. Let *u* be **TimeClip**(**UTC**(**MakeDate**(**Day**(*t*), *time*))).
5. Set the [[DateValue]] internal slot of **this** **Date** object to *u*.
6. Return *u*.

21.4.4.24 Date.prototype.setMinutes (*min* [, *sec* [, *ms*]])

The following steps are performed:

1. Let *t* be **LocalTime**(? **thisTimeValue**(**this** value)).
2. Let *m* be ? **ToNumber**(*min*).
3. If *sec* is not present, let *s* be **SecFromTime**(*t*); otherwise, let *s* be ? **ToNumber**(*sec*).
4. If *ms* is not present, let *milli* be **msFromTime**(*t*); otherwise, let *milli* be ? **ToNumber**(*ms*).
5. Let *date* be **MakeDate**(**Day**(*t*), **MakeTime**(**HourFromTime**(*t*), *m*, *s*, *milli*)).
6. Let *u* be **TimeClip**(**UTC**(*date*)).
7. Set the [[DateValue]] internal slot of **this** **Date** object to *u*.
8. Return *u*.

The "length" property of the **setMinutes** method is 3_F.

NOTE If *sec* is not present, this method behaves as if *sec* was present with the value **getSeconds()**. If *ms* is not present, this behaves as if *ms* was present with the value **getMilliseconds()**.

21.4.4.25 Date.prototype.setMonth (*month* [, *date*])

The following steps are performed:

1. Let *t* be **LocalTime**(? **thisTimeValue**(**this** value)).
2. Let *m* be ? **ToNumber**(*month*).
3. If *date* is not present, let *dt* be **DateFromTime**(*t*); otherwise, let *dt* be ? **ToNumber**(*date*).
4. Let *newDate* be **MakeDate**(**MakeDay**(**YearFromTime**(*t*), *m*, *dt*), **TimeWithinDay**(*t*)).
5. Let *u* be **TimeClip**(**UTC**(*newDate*)).
6. Set the [[DateValue]] internal slot of **this** **Date** object to *u*.
7. Return *u*.

The "length" property of the **setMonth** method is 2_F.

NOTE If *date* is not present, this method behaves as if *date* was present with the value **getDate()**.

21.4.4.26 Date.prototype.setSeconds (*sec* [, *ms*])

The following steps are performed:

1. Let *t* be **LocalTime**(? **thisTimeValue**(**this** value)).
2. Let *s* be ? **ToNumber**(*sec*).
3. If *ms* is not present, let *milli* be **msFromTime**(*t*); otherwise, let *milli* be ? **ToNumber**(*ms*).
4. Let *date* be **MakeDate**(**Day**(*t*), **MakeTime**(**HourFromTime**(*t*), **MinFromTime**(*t*), *s*, *milli*)).
5. Let *u* be **TimeClip**(**UTC**(*date*)).
6. Set the [[DateValue]] internal slot of **this Date object** to *u*.
7. Return *u*.

The "length" property of the **setSeconds** method is 2_F.

NOTE If *ms* is not present, this method behaves as if *ms* was present with the value **getMilliseconds()**.

21.4.4.27 Date.prototype.setTime (*time*)

The following steps are performed:

1. Perform ? **thisTimeValue**(**this** value).
2. Let *t* be ? **ToNumber**(*time*).
3. Let *v* be **TimeClip**(*t*).
4. Set the [[DateValue]] internal slot of **this Date object** to *v*.
5. Return *v*.

21.4.4.28 Date.prototype.setUTCDate (*date*)

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *dt* be ? **ToNumber**(*date*).
3. Let *newDate* be **MakeDate**(**MakeDay**(**YearFromTime**(*t*), **MonthFromTime**(*t*), *dt*), **TimeWithinDay**(*t*)).
4. Let *v* be **TimeClip**(*newDate*).
5. Set the [[DateValue]] internal slot of **this Date object** to *v*.
6. Return *v*.

21.4.4.29 Date.prototype.setUTCFullYear (*year* [, *month* [, *date*]])

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. If *t* is NaN, set *t* to +0_F.

3. Let *y* be ? **ToNumber**(*year*).
4. If *month* is not present, let *m* be **MonthFromTime**(*t*); otherwise, let *m* be ? **ToNumber**(*month*).
5. If *date* is not present, let *dt* be **DateFromTime**(*t*); otherwise, let *dt* be ? **ToNumber**(*date*).
6. Let *newDate* be **MakeDate**(**MakeDay**(*y*, *m*, *dt*), **TimeWithinDay**(*t*)).
7. Let *v* be **TimeClip**(*newDate*).
8. Set the [[DateValue]] internal slot of **this Date object** to *v*.
9. Return *v*.

The "length" property of the **setUTCFullYear** method is 3_F.

NOTE If *month* is not present, this method behaves as if *month* was present with the value **getUTCMonth()**. If *date* is not present, it behaves as if *date* was present with the value **getUTCDate()**.

21.4.4.30 Date.prototype.setUTCHours (*hour* [, *min* [, *sec* [, *ms*]]])

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *h* be ? **ToNumber**(*hour*).
3. If *min* is not present, let *m* be **MinFromTime**(*t*); otherwise, let *m* be ? **ToNumber**(*min*).
4. If *sec* is not present, let *s* be **SecFromTime**(*t*); otherwise, let *s* be ? **ToNumber**(*sec*).
5. If *ms* is not present, let *milli* be **msFromTime**(*t*); otherwise, let *milli* be ? **ToNumber**(*ms*).
6. Let *newDate* be **MakeDate**(**Day**(*t*), **MakeTime**(*h*, *m*, *s*, *milli*)).
7. Let *v* be **TimeClip**(*newDate*).
8. Set the [[DateValue]] internal slot of **this Date object** to *v*.
9. Return *v*.

The "length" property of the **setUTCHours** method is 4_F.

NOTE If *min* is not present, this method behaves as if *min* was present with the value **getUTCMinutes()**. If *sec* is not present, it behaves as if *sec* was present with the value **getUTCSeconds()**. If *ms* is not present, it behaves as if *ms* was present with the value **getUTCMilliseconds()**.

21.4.4.31 Date.prototype.setUTCMilliseconds (*ms*)

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *milli* be ? **ToNumber**(*ms*).
3. Let *time* be **MakeTime**(**HourFromTime**(*t*), **MinFromTime**(*t*), **SecFromTime**(*t*), *milli*).
4. Let *v* be **TimeClip**(**MakeDate**(**Day**(*t*), *time*)).
5. Set the [[DateValue]] internal slot of **this Date object** to *v*.
6. Return *v*.

21.4.4.32 Date.prototype.setUTCMinutes (*min* [, *sec* [, *ms*]])

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *m* be ? **ToNumber**(*min*).
3. If *sec* is not present, let *s* be **SecFromTime**(*t*).
4. Else,
 - a. Let *s* be ? **ToNumber**(*sec*).
5. If *ms* is not present, let *milli* be **msFromTime**(*t*).
6. Else,
 - a. Let *milli* be ? **ToNumber**(*ms*).
7. Let *date* be **MakeDate**(**Day**(*t*), **MakeTime**(**HourFromTime**(*t*), *m*, *s*, *milli*)).
8. Let *v* be **TimeClip**(*date*).
9. Set the [[DateValue]] internal slot of **this Date object** to *v*.
10. Return *v*.

The "length" property of the **setUTCMinutes** method is 3_F.

NOTE If *sec* is not present, this method behaves as if *sec* was present with the value **getUTCSeconds()**. If *ms* is not present, it function behaves as if *ms* was present with the value return by **getUTCMilliseconds()**.

21.4.4.33 Date.prototype.setUTCMonth (*month* [, *date*])

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *m* be ? **ToNumber**(*month*).
3. If *date* is not present, let *dt* be **DateFromTime**(*t*).
4. Else,
 - a. 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_F.

NOTE If *date* is not present, this method behaves as if *date* was present with the value **getUTCDate()**.

21.4.4.34 Date.prototype.setUTCSeconds (*sec* [, *ms*])

The following steps are performed:

1. Let *t* be ? **thisTimeValue**(**this** value).
2. Let *s* be ? **ToNumber**(*sec*).
3. If *ms* is not present, let *milli* be **msFromTime**(*t*).
4. Else,

- a. Let *milli* be ? `ToNumber(ms)`.
5. Let *date* be `MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli))`.
6. Let *v* be `TimeClip(date)`.
7. Set the `[[DateValue]]` internal slot of this Date object to *v*.
8. Return *v*.

The "length" property of the `setUTCSeconds` method is 2.

NOTE If *ms* is not present, this method behaves as if *ms* was present with the value `getUTCMilliseconds()`.

21.4.4.35 Date.prototype.toString ()

The following steps are performed:

1. Let *O* be this Date object.
2. Let *tv* be ? `thisTimeValue(O)`.
3. If *tv* is NaN, return "Invalid Date".
4. Let *t* be `LocalTime(tv)`.
5. Return `DateString(t)`.

21.4.4.36 Date.prototype.toISOString ()

If this time value is not a finite Number or if it corresponds with a year that cannot be represented in the Date Time String Format, this function throws a `RangeError` exception. Otherwise, it returns a String representation of this time value in that format on the UTC time scale, including all format elements and the UTC offset representation "Z".

21.4.4.37 Date.prototype.toJSON (key)

This function provides a String representation of a Date object for use by `JSON.stringify` (25.5.2).

When the `toJSON` method is called with argument *key*, the following steps are taken:

1. Let *O* be ? `ToObject(this value)`.
2. Let *tv* be ? `ToPrimitive(O, number)`.
3. If `Type(tv)` is Number and *tv* is not finite, return `null`.
4. Return ? `Invoke(O, "toISOString")`.

NOTE 1 The argument is ignored.

NOTE 2 The `toJSON` function is intentionally generic; it does not require that its `this` value be a Date object. Therefore, it can be transferred to other kinds of objects for use as a method. However, it does require that any such object have a `toISOString` method.

21.4.4.38 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-defined](#), 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.

21.4.4.39 Date.prototype.toLocaleString ([*reserved1* [, *reserved2*]])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the **Date.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.

This function returns a String value. The contents of the String are [implementation-defined](#), 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.

21.4.4.40 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-defined](#), 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.

21.4.4.41 Date.prototype.toString ()

The following steps are performed:

1. Let *tv* be ? [thisTimeValue](#)(**this** value).
2. Return [ToDateToString](#)(*tv*).

NOTE 1

For any Date object **d** such that **d**.[\[\[DateValue\]\]](#) is evenly divisible by 1000, the result of **Date.parse(d.toString())** = **d.valueOf()**. See [21.4.3.2](#).

The **toString** function is not generic; it throws a **TypeError** exception if its **this** value is not a Date object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

21.4.4.41.1 TimeString (*tv*)

The abstract operation TimeString takes argument *tv*. It performs the following steps when called:

1. Assert: `Type(tv)` is Number.
2. Assert: *tv* is not NaN.
3. Let *hour* be the String representation of `HourFromTime(tv)`, formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
4. Let *minute* be the String representation of `MinFromTime(tv)`, formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
5. Let *second* be the String representation of `SecFromTime(tv)`, formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
6. Return the string-concatenation of *hour*, ":", *minute*, ":", *second*, the code unit 0x0020 (SPACE), and "GMT".

21.4.4.41.2 DateString (*tv*)

The abstract operation DateString takes argument *tv*. It performs the following steps when called:

1. Assert: `Type(tv)` is Number.
2. Assert: *tv* is not NaN.
3. Let *weekday* be the Name of the entry in Table 52 with the Number `WeekDay(tv)`.
4. Let *month* be the Name of the entry in Table 53 with the Number `MonthFromTime(tv)`.
5. Let *day* be the String representation of `DateFromTime(tv)`, formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
6. Let *yv* be `YearFromTime(tv)`.
7. If $yv \geq +0_{\mathbb{F}}$, let *yearSign* be the empty String; otherwise, let *yearSign* be "-".
8. Let *year* be the String representation of `abs($\mathbb{R}(yv)$)`, formatted as a decimal number.
9. Let *paddedYear* be `!StringPad(year, 4 \mathbb{F} , "0", start)`.
10. Return the string-concatenation of *weekday*, the code unit 0x0020 (SPACE), *month*, the code unit 0x0020 (SPACE), *day*, the code unit 0x0020 (SPACE), *yearSign*, and *paddedYear*.

Table 52: Names of days of the week

Number	Name
+0 _{\mathbb{F}}	"Sun"
1 _{\mathbb{F}}	"Mon"
2 _{\mathbb{F}}	"Tue"
3 _{\mathbb{F}}	"Wed"
4 _{\mathbb{F}}	"Thu"
5 _{\mathbb{F}}	"Fri"
6 _{\mathbb{F}}	"Sat"

Table 53: Names of months of the year

Number	Name
+0 _F	"Jan"
1 _F	"Feb"
2 _F	"Mar"
3 _F	"Apr"
4 _F	"May"
5 _F	"Jun"
6 _F	"Jul"
7 _F	"Aug"
8 _F	"Sep"
9 _F	"Oct"
10 _F	"Nov"
11 _F	"Dec"

21.4.4.41.3 TimeZoneString (*tv*)

The abstract operation TimeZoneString takes argument *tv*. It performs the following steps when called:

1. **Assert:** Type(*tv*) is Number.
2. **Assert:** *tv* is not NaN.
3. Let *offset* be LocalTZA(*tv*, true).
4. If *offset* ≥ +0_F, then
 - a. Let *offsetSign* be "+".
 - b. Let *absOffset* be *offset*.
5. Else,
 - a. Let *offsetSign* be "-".
 - b. Let *absOffset* be -*offset*.
6. Let *offsetMin* be the String representation of MinFromTime(*absOffset*), formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
7. Let *offsetHour* be the String representation of HourFromTime(*absOffset*), formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
8. Let *tzName* be an implementation-defined string that is either the empty String or the string-concatenation of the code unit 0x0020 (SPACE), the code unit 0x0028 (LEFT PARENTHESIS), an implementation-defined timezone name, and the code unit 0x0029 (RIGHT PARENTHESIS).
9. Return the string-concatenation of *offsetSign*, *offsetHour*, *offsetMin*, and *tzName*.

21.4.4.41.4 ToDateString (*tv*)

The abstract operation `ToDateString` takes argument *tv*. It performs the following steps when called:

1. **Assert:** `Type(tv)` is `Number`.
2. If *tv* is `NaN`, return `"Invalid Date"`.
3. Let *t* be `LocalTime(tv)`.
4. Return the **string-concatenation** of `DateString(t)`, the code unit 0x0020 (SPACE), `TimeString(t)`, and `TimeZoneString(tv)`.

21.4.4.42 `Date.prototype.toTimeString` ()

The following steps are performed:

1. Let *O* be **this** `Date` object.
2. Let *tv* be ? `thisTimeValue(O)`.
3. If *tv* is `NaN`, return `"Invalid Date"`.
4. Let *t* be `LocalTime(tv)`.
5. Return the **string-concatenation** of `TimeString(t)` and `TimeZoneString(tv)`.

21.4.4.43 `Date.prototype.toUTCString` ()

The **`toUTCString`** method returns a `String` value representing the instance in time corresponding to **this time value**. The format of the `String` is based upon "HTTP-date" from RFC 7231, generalized to support the full range of times supported by ECMAScript `Date` objects. It performs the following steps when called:

1. Let *O* be **this** `Date` object.
2. Let *tv* be ? `thisTimeValue(O)`.
3. If *tv* is `NaN`, return `"Invalid Date"`.
4. Let *weekday* be the Name of the entry in [Table 52](#) with the Number `WeekDay(tv)`.
5. Let *month* be the Name of the entry in [Table 53](#) with the Number `MonthFromTime(tv)`.
6. Let *day* be the `String` representation of `DateFromTime(tv)`, formatted as a two-digit decimal number, padded to the left with the code unit 0x0030 (DIGIT ZERO) if necessary.
7. Let *yv* be `YearFromTime(tv)`.
8. If $yv \geq +0_{\mathbb{F}}$, let *yearSign* be the empty `String`; otherwise, let *yearSign* be `"-"`.
9. Let *year* be the `String` representation of `abs($\mathbb{R}(yv)$)`, formatted as a decimal number.
10. Let *paddedYear* be ! `StringPad(year, 4F, "0", start)`.
11. Return the **string-concatenation** of *weekday*, `","`, the code unit 0x0020 (SPACE), *day*, the code unit 0x0020 (SPACE), *month*, the code unit 0x0020 (SPACE), *yearSign*, *paddedYear*, the code unit 0x0020 (SPACE), and `TimeString(tv)`.

21.4.4.44 `Date.prototype.valueOf` ()

The following steps are performed:

1. Return ? `thisTimeValue(this value)`.

21.4.4.45 `Date.prototype [@@toPrimitive] (hint)`

This function is called by ECMAScript language operators to convert a `Date` 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, throw a **TypeError** exception.
3. If *hint* is **"string"** or **"default"**, then
 - a. Let *tryFirst* be string.
4. Else if *hint* is **"number"**, then
 - a. Let *tryFirst* be number.
5. Else, throw a **TypeError** exception.
6. Return ? **OrdinaryToPrimitive**(*O*, *tryFirst*).

The value of the **"name"** property of this function is **"[Symbol.toPrimitive]"**.

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

21.4.5 Properties of Date Instances

Date instances are ordinary objects that inherit properties from the **Date prototype object**. Date instances also have a **[[DateValue]]** internal slot. The **[[DateValue]]** internal slot is the **time value** represented by **this Date object**.

22 Text Processing

22.1 String Objects

22.1.1 The String Constructor

The String **constructor**:

- is *%String%*.
- is the initial value of the **"String"** property of the **global object**.
- creates and initializes a new String object when called as a **constructor**.
- performs a type conversion when called as a function rather than as a **constructor**.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified String behaviour must include a **super** call to the String **constructor** to create and initialize the subclass instance with a **[[StringData]]** internal slot.

22.1.1.1 String (*value*)

When **String** is called with argument *value*, the following steps are taken:

1. If *value* is not present, let *s* be the empty String.
2. Else,
 - a. If **NewTarget** is **undefined** and **Type**(*value*) is Symbol, return **SymbolDescriptiveString**(*value*).

- b. Let s be `? ToString(value)`.
3. If `NewTarget` is **undefined**, return s .
4. Return `! StringCreate(s, ? GetPrototypeFromConstructor(NewTarget, "%String.prototype%"))`.

22.1.2 Properties of the String Constructor

The String `constructor`:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

22.1.2.1 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 $length$ be the number of elements in `codeUnits`.
2. Let $elements$ be a new empty `List`.
3. For each element $next$ of `codeUnits`, do
 - a. Let $nextCU$ be `ℝ(? ToUint16(next))`.
 - b. Append $nextCU$ to the end of $elements$.
4. Return the String value whose code units are the elements in the `List elements`. If `codeUnits` is empty, the empty String is returned.

The "length" property of the **fromCharCode** function is `1F`.

22.1.2.2 String.fromCharCodePoint (...codePoints)

The **String.fromCharCodePoint** function may be called with any number of arguments which form the rest parameter `codePoints`. The following steps are taken:

1. Let $result$ be the empty String.
2. For each element $next$ of `codePoints`, do
 - a. Let $nextCP$ be `? ToNumber(next)`.
 - b. If `! IsIntegralNumber(nextCP)` is **false**, throw a **RangeError** exception.
 - c. If `ℝ(nextCP) < 0` or `ℝ(nextCP) > 0x10FFFF`, throw a **RangeError** exception.
 - d. Set $result$ to the `string-concatenation` of $result$ and `! UTF16EncodeCodePoint(ℝ(nextCP))`.
3. **Assert**: If `codePoints` is empty, then $result$ is the empty String.
4. Return $result$.

The "length" property of the **fromCodePoint** function is `1F`.

22.1.2.3 String.prototype

The initial value of **String.prototype** is the `String prototype` object.

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

22.1.2.4 String.raw (template, ...substitutions)

The **String.raw** function may be called with a variable number of arguments. The first argument is *template* and the remainder of the arguments form the *List substitutions*. The following steps are taken:

1. Let *numberOfSubstitutions* be the number of elements in *substitutions*.
2. Let *cooked* be ? **ToObject**(*template*).
3. Let *raw* be ? **ToObject**(? **Get**(*cooked*, "raw")).
4. Let *literalSegments* be ? **LengthOfArrayLike**(*raw*).
5. If *literalSegments* ≤ 0, return the empty String.
6. Let *stringElements* be a new empty **List**.
7. Let *nextIndex* be 0.
8. Repeat,
 - a. Let *nextKey* be ! **ToString**(**F**(*nextIndex*)).
 - b. Let *nextSeg* be ? **ToString**(? **Get**(*raw*, *nextKey*)).
 - c. Append the code unit elements of *nextSeg* to the end of *stringElements*.
 - d. If *nextIndex* + 1 = *literalSegments*, then
 - i. Return the String value whose code units are the elements in the **List** *stringElements*. If *stringElements* has no elements, the empty String is returned.
 - e. If *nextIndex* < *numberOfSubstitutions*, let *next* be *substitutions*[*nextIndex*].
 - f. Else, let *next* be the empty String.
 - g. Let *nextSub* be ? **ToString**(*next*).
 - h. Append the code unit elements of *nextSub* to the end of *stringElements*.
 - i. Set *nextIndex* to *nextIndex* + 1.

NOTE

The **raw** function is intended for use as a tag function of a Tagged Template (13.3.11). When called as such, the first argument will be a well formed template object and the rest parameter will contain the substitution values.

22.1.3 Properties of the String Prototype Object

The *String prototype object*:

- is %*String.prototype*%.
- is a **String exotic object** and has the internal methods specified for such objects.
- has a **[[StringData]]** internal slot whose value is the empty String.
- has a **"length"** property whose initial value is +0_F and whose attributes are { **[[Writable]]**: **false**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **false** }.
- has a **[[Prototype]]** internal slot whose value is %*Object.prototype*%.

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* takes argument *value*. It performs the following steps when called:

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 *value*.**[[StringData]]**.
 - b. **Assert**: **Type**(*s*) is String.

- c. Return *s*.
3. Throw a **TypeError** exception.

22.1.3.1 String.prototype.charAt (*pos*)

NOTE 1 Returns a single element String containing the code unit at index *pos* within the String value resulting from converting this object to a String. If there is no element at that index, the result is the empty String. The result is a String value, not a String object.

If **pos** is an **integral Number**, then the result of **x.charAt(pos)** is equivalent 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. Let *position* be ? **ToIntegerOrInfinity**(*pos*).
4. Let *size* be the length of *S*.
5. If *position* < 0 or *position* ≥ *size*, return the empty String.
6. Return the String value of length 1, containing one code unit from *S*, namely the code unit at index *position*.

NOTE 2 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.

22.1.3.2 String.prototype.charCodeAt (*pos*)

NOTE 1 Returns a Number (a non-negative **integral Number** less than 2^{16}) that is the numeric value of the code unit at index *pos* within the String resulting from converting this object to a String. If there is no element at that index, 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. Let *position* be ? **ToIntegerOrInfinity**(*pos*).
4. Let *size* be the length of *S*.
5. If *position* < 0 or *position* ≥ *size*, return **NaN**.
6. Return the **Number value** for the numeric value of the code unit at index *position* within the String *S*.

NOTE 2 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.

22.1.3.3 String.prototype.codePointAt (*pos*)

NOTE 1 Returns a non-negative **integral Number** less than or equal to $0x10FFFF_{\mathbb{F}}$ that is the code point value of the UTF-16 encoded code point (6.1.4) starting at the string element at index *pos* within the String resulting from converting this object to a String. If there is no element at that index, 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. Let *position* be ? **ToIntegerOrInfinity**(*pos*).
4. Let *size* be the length of *S*.
5. If *position* < 0 or *position* ≥ *size*, return **undefined**.
6. Let *cp* be ! **CodePointAt**(*S*, *position*).
7. Return $\mathbb{F}(cp.[[CodePoint]])$.

NOTE 2 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.

22.1.3.4 String.prototype.concat (...args)

NOTE 1 When the **concat** method is called it returns the String value consisting of the code units of the **this** value (converted to a String) followed by the code units 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. Let *R* be *S*.
4. For each element *next* of *args*, do
 - a. Let *nextString* be ? **ToString**(*next*).
 - b. Set *R* to the **string-concatenation** of *R* and *nextString*.
5. Return *R*.

The "length" property of the **concat** method is $1_{\mathbb{F}}$.

NOTE 2 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.

22.1.3.5 String.prototype.constructor

The initial value of **String.prototype.constructor** is *%String%*.

22.1.3.6 String.prototype.endsWith (searchString [, endPosition])

The following steps are taken:

1. Let *O* be ? `RequireObjectCoercible`(**this** value).
2. Let *S* be ? `ToString`(*O*).
3. Let *isRegExp* be ? `IsRegExp`(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? `ToString`(*searchString*).
6. Let *len* be the length of *S*.
7. If *endPosition* is **undefined**, let *pos* be *len*; else let *pos* be ? `ToIntegerOrInfinity`(*endPosition*).
8. Let *end* be the result of `clamping` *pos* between 0 and *len*.
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength* = 0, return **true**.
11. Let *start* be *end* - *searchLength*.
12. If *start* < 0, return **false**.
13. Let *substring* be the `substring` of *S* from *start* to *end*.
14. Return ! `SameValueNonNumeric`(*substring*, *searchStr*).

NOTE 1 Returns **true** if the sequence of code units of *searchString* converted to a String is the same as the corresponding code units 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 extensions 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.

22.1.3.7 `String.prototype.includes` (*searchString* [, *position*])

The **includes** 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. Let *isRegExp* be ? `IsRegExp`(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? `ToString`(*searchString*).
6. Let *pos* be ? `ToIntegerOrInfinity`(*position*).
7. **Assert**: If *pos* is **undefined**, then *pos* is 0.
8. Let *len* be the length of *S*.
9. Let *start* be the result of `clamping` *pos* between 0 and *len*.
10. Let *index* be ! `StringIndexOf`(*S*, *searchStr*, *start*).
11. If *index* is not -1, return **true**.
12. Return **false**.

NOTE 1 If *searchString* appears as a substring of the result of converting this object to a String, at one or more indices that are greater than or equal to *position*, 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 **includes** 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.

22.1.3.8 String.prototype.indexOf (*searchString* [, *position*])

NOTE 1 If *searchString* appears as a substring of the result of converting this object to a String, at one or more indices that are greater than or equal to *position*, then the smallest such index is returned; otherwise, **-1**_F is returned. If *position* is **undefined**, **+0**_F 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. Let *searchStr* be ? **ToString**(*searchString*).
4. Let *pos* be ? **ToIntegerOrInfinity**(*position*).
5. **Assert**: If *position* is **undefined**, then *pos* is 0.
6. Let *len* be the length of *S*.
7. Let *start* be the result of **clamping** *pos* between 0 and *len*.
8. Return **F**(! **StringIndexOf**(*S*, *searchStr*, *start*)).

NOTE 2 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.

22.1.3.9 String.prototype.lastIndexOf (*searchString* [, *position*])

NOTE 1 If *searchString* appears as a substring of the result of converting this object to a String at one or more indices that are smaller than or equal to *position*, then the greatest such index is returned; otherwise, **-1**_F 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. Let *searchStr* be ? **ToString**(*searchString*).
4. Let *numPos* be ? **ToNumber**(*position*).
5. **Assert**: If *position* is **undefined**, then *numPos* is NaN.

6. If *numPos* is **NaN**, let *pos* be $+\infty$; otherwise, let *pos* be `! ToIntegerOrInfinity(numPos)`.
7. Let *len* be the length of *S*.
8. Let *start* be the result of `clamping pos` between 0 and *len*.
9. Let *searchLen* be the length of *searchStr*.
10. Let *k* be the largest possible non-negative **integer** not larger than *start* such that $k + \textit{searchLen} \leq \textit{len}$, and for all non-negative integers *j* such that $j < \textit{searchLen}$, the code unit at index $k + j$ within *S* is the same as the code unit at index *j* within *searchStr*; but if there is no such **integer**, let *k* be -1.
11. Return `F(k)`.

NOTE 2 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.

22.1.3.10 `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 performing 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. Let *That* be `? ToString(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 23.1.3.27) 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.

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 comparison capabilities, this function must treat Strings that are canonically equivalent according to the Unicode standard as identical. It is recommended that this function should 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 Standard Annex #15, Unicode Normalization Forms (<https://unicode.org/reports/tr15/>) and Unicode Technical Note #5, Canonical Equivalence in Applications (<https://www.unicode.org/notes/tn5/>). Also see Unicode Technical Standard #10, Unicode Collation Algorithm (<https://unicode.org/reports/tr10/>).

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.

22.1.3.11 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. If *regexp* is neither **undefined** nor **null**, then
 - a. Let *matcher* be ? **GetMethod**(*regexp*, @@**match**).
 - b. If *matcher* is not **undefined**, then
 - i. Return ? **Call**(*matcher*, *regexp*, « *O* »).
3. Let *S* be ? **ToString**(*O*).
4. Let *rx* be ? **RegExpCreate**(*regexp*, **undefined**).
5. Return ? **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.

22.1.3.12 String.prototype.matchAll (*regexp*)

Performs a regular expression match of the String representing the **this** value against *regexp* and returns an iterator. Each iteration result's value is an Array object containing the results of the match, or **null** if the String did not match.

When the **matchAll** method is called, the following steps are taken:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. If *regexp* is neither **undefined** nor **null**, then
 - a. Let *isRegExp* be ? **IsRegExp**(*regexp*).
 - b. If *isRegExp* is **true**, then
 - i. Let *flags* be ? **Get**(*regexp*, "flags").
 - ii. Perform ? **RequireObjectCoercible**(*flags*).
 - iii. If ? **ToString**(*flags*) does not contain "g", throw a **TypeError** exception.
 - c. Let *matcher* be ? **GetMethod**(*regexp*, @@**matchAll**).

- d. If *matcher* is not **undefined**, then
 - i. Return ? **Call**(*matcher*, *regexp*, « *O* »).
3. Let *S* be ? **ToString**(*O*).
4. Let *rx* be ? **RegExpCreate**(*regexp*, "g").
5. Return ? **Invoke**(*rx*, @@**matchAll**, « *S* »).

NOTE 1 The **matchAll** 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.

NOTE 2 Similarly to **String.prototype.split**, **String.prototype.matchAll** is designed to typically act without mutating its inputs.

22.1.3.13 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. If *form* is **undefined**, let *f* be "NFC".
4. Else, let *f* be ? **ToString**(*form*).
5. If *f* is not one of "NFC", "NFD", "NFKC", or "NFKD", throw a **RangeError** exception.
6. Let *ns* be the String value that is the result of normalizing *S* into the normalization form named by *f* as specified in <https://unicode.org/reports/tr15/>.
7. Return *ns*.

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.

22.1.3.14 String.prototype.padEnd (*maxLength* [, *fillString*])

When the **padEnd** method is called, the following steps are taken:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. Return ? **StringPad**(*O*, *maxLength*, *fillString*, end).

22.1.3.15 String.prototype.padStart (*maxLength* [, *fillString*])

When the **padStart** method is called, the following steps are taken:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. Return ? **StringPad**(*O*, *maxLength*, *fillString*, start).

22.1.3.15.1 StringPad (*O*, *maxLength*, *fillString*, *placement*)

The abstract operation **StringPad** takes arguments *O*, *maxLength*, *fillString*, and *placement*. It performs the following steps when called:

1. **Assert:** *placement* is start or end.
2. Let *S* be ? `ToString(O)`.
3. Let *intMaxLength* be \mathbb{R} (? `ToLength(maxLength)`).
4. Let *stringLength* be the length of *S*.
5. If $intMaxLength \leq stringLength$, return *S*.
6. If *fillString* is **undefined**, let *filler* be the String value consisting solely of the code unit 0x0020 (SPACE).
7. Else, let *filler* be ? `ToString(fillString)`.
8. If *filler* is the empty String, return *S*.
9. Let *fillLen* be $intMaxLength - stringLength$.
10. Let *truncatedStringFiller* be the String value consisting of repeated concatenations of *filler* truncated to length *fillLen*.
11. If *placement* is **start**, return the **string-concatenation** of *truncatedStringFiller* and *S*.
12. Else, return the **string-concatenation** of *S* and *truncatedStringFiller*.

NOTE 1 The argument *maxLength* will be clamped such that it can be no smaller than the length of *S*.

NOTE 2 The argument *fillString* defaults to "" (the String value consisting of the code unit 0x0020 SPACE).

22.1.3.16 String.prototype.repeat (*count*)

The following steps are taken:

1. Let *O* be ? `RequireObjectCoercible(this value)`.
2. Let *S* be ? `ToString(O)`.
3. Let *n* be ? `ToIntegerOrInfinity(count)`.
4. If $n < 0$ or *n* is $+\infty$, throw a **RangeError** exception.
5. If *n* is 0, return the empty String.
6. Return the String value that is made from *n* copies of *S* appended together.

NOTE 1 This method creates the String value consisting of the code units of the **this** value (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.

22.1.3.17 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. If *searchValue* is neither **undefined** nor **null**, then
 - a. Let *replacer* be ? `GetMethod(searchValue, @@replace)`.
 - b. If *replacer* is not **undefined**, then
 - i. Return ? `Call(replacer, searchValue, « O, replaceValue »)`.

3. Let *string* be ? ToString(*O*).
4. Let *searchString* be ? ToString(*searchValue*).
5. Let *functionalReplace* be IsCallable(*replaceValue*).
6. If *functionalReplace* is **false**, then
 - a. Set *replaceValue* to ? ToString(*replaceValue*).
7. Let *searchLength* be the length of *searchString*.
8. Let *position* be ! StringIndexOf(*string*, *searchString*, 0).
9. If *position* is -1, return *string*.
10. Let *preserved* be the substring of *string* from 0 to *position*.
11. If *functionalReplace* is **true**, then
 - a. Let *replacement* be ? ToString(? Call(*replaceValue*, **undefined**, « *searchString*, F(*position*), *string* »)).
12. Else,
 - a. Assert: Type(*replaceValue*) is String.
 - b. Let *captures* be a new empty List.
 - c. Let *replacement* be ! GetSubstitution(*searchString*, *string*, *position*, *captures*, **undefined**, *replaceValue*).
13. Return the string-concatenation of *preserved*, *replacement*, and the substring of *string* from *position* + *searchLength*.

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.

22.1.3.17.1 GetSubstitution (*matched*, *str*, *position*, *captures*, *namedCaptures*, *replacement*)

The abstract operation GetSubstitution takes arguments *matched*, *str*, *position* (a non-negative integer), *captures*, *namedCaptures*, and *replacement*. It performs the following steps when called:

1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matched*.
3. Assert: Type(*str*) is String.
4. Let *stringLength* be the number of code units in *str*.
5. Assert: *position* ≤ *stringLength*.
6. Assert: *captures* is a possibly empty List of Strings.
7. Assert: Type(*replacement*) is String.
8. Let *tailPos* be *position* + *matchLength*.
9. Let *m* be the number of elements in *captures*.
10. Let *result* be the String value derived from *replacement* by copying code unit elements from *replacement* to *result* while performing replacements as specified in Table 54. These \$ replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements.
11. Return *result*.

Table 54: Replacement Text Symbol Substitutions

Code units	Unicode Characters	Replacement text
0x0024, 0x0024	\$\$	\$
0x0024, 0x0026	\$&	<i>matched</i>
0x0024, 0x0060	\$`	The replacement is the substring of <i>str</i> from 0 to <i>position</i> .
0x0024, 0x0027	\$'	If <i>tailPos</i> \geq <i>stringLength</i> , the replacement is the empty String. Otherwise the replacement is the substring of <i>str</i> from <i>tailPos</i> .
0x0024, N Where 0x0031 $\leq N \leq$ 0x0039	\$n where n is one of 1 2 3 4 5 6 7 8 9 and \$n is not followed by a decimal digit	The n^{th} element of <i>captures</i> , where <i>n</i> is a single digit in the range 1 to 9. If $n \leq m$ and the n^{th} element of <i>captures</i> is undefined , use the empty String instead. If $n > m$, no replacement is done.
0x0024, N, N Where 0x0030 $\leq N \leq$ 0x0039	\$nn where n is one of 0 1 2 3 4 5 6 7 8 9	The nn^{th} element of <i>captures</i> , where <i>nn</i> is a two-digit decimal number in the range 01 to 99. If $nn \leq m$ and the nn^{th} element of <i>captures</i> is undefined , use the empty String instead. If <i>nn</i> is 00 or $nn > m$, no replacement is done.
0x0024, 0x003C	\$<	<ol style="list-style-type: none"> 1. If <i>namedCaptures</i> is undefined, the replacement text is the String "\$<". 2. Else, <ol style="list-style-type: none"> a. Assert: <i>Type(namedCaptures)</i> is Object. b. Scan until the next > U+003E (GREATER-THAN SIGN). c. If none is found, the replacement text is the String "\$<". d. Else, <ol style="list-style-type: none"> i. Let <i>groupName</i> be the enclosed substring. ii. Let <i>capture</i> be ? <i>Get(namedCaptures, groupName)</i>. iii. If <i>capture</i> is undefined, replace the text through > with the empty String. iv. Otherwise, replace the text through > with ? <i>Tostring(capture)</i>.
0x0024	\$ in any context that does not match any of the above.	\$

22.1.3.18 String.prototype.replaceAll (*searchValue*, *replaceValue*)

When the **replaceAll** method is called with arguments *searchValue* and *replaceValue*, the following steps are taken:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. If *searchValue* is neither **undefined** nor **null**, then
 - a. Let *isRegExp* be ? **IsRegExp**(*searchValue*).
 - b. If *isRegExp* is **true**, then
 - i. Let *flags* be ? **Get**(*searchValue*, "flags").
 - ii. Perform ? **RequireObjectCoercible**(*flags*).
 - iii. If ? **Tostring**(*flags*) does not contain "g", throw a **TypeError** exception.
 - c. Let *replacer* be ? **GetMethod**(*searchValue*, @@replace).
 - d. If *replacer* is not **undefined**, then
 - i. Return ? **Call**(*replacer*, *searchValue*, « *O*, *replaceValue* »).
3. Let *string* be ? **Tostring**(*O*).
4. Let *searchString* be ? **Tostring**(*searchValue*).
5. Let *functionalReplace* be **IsCallable**(*replaceValue*).
6. If *functionalReplace* is **false**, then
 - a. Set *replaceValue* to ? **Tostring**(*replaceValue*).
7. Let *searchLength* be the length of *searchString*.
8. Let *advanceBy* be **max**(1, *searchLength*).
9. Let *matchPositions* be a new empty **List**.
10. Let *position* be ! **StringIndexOf**(*string*, *searchString*, 0).
11. Repeat, while *position* is not -1,
 - a. Append *position* to the end of *matchPositions*.
 - b. Set *position* to ! **StringIndexOf**(*string*, *searchString*, *position* + *advanceBy*).
12. Let *endOfLastMatch* be 0.
13. Let *result* be the empty **String**.
14. For each element *p* of *matchPositions*, do
 - a. Let *preserved* be the **substring** of *string* from *endOfLastMatch* to *p*.
 - b. If *functionalReplace* is **true**, then
 - i. Let *replacement* be ? **Tostring**(? **Call**(*replaceValue*, **undefined**, « *searchString*, **F**(*p*), *string* »)).
 - c. Else,
 - i. **Assert**: **Type**(*replaceValue*) is **String**.
 - ii. Let *captures* be a new empty **List**.
 - iii. Let *replacement* be ! **GetSubstitution**(*searchString*, *string*, *p*, *captures*, **undefined**, *replaceValue*).
 - d. Set *result* to the **string-concatenation** of *result*, *preserved*, and *replacement*.
 - e. Set *endOfLastMatch* to *p* + *searchLength*.
15. If *endOfLastMatch* < the length of *string*, then
 - a. Set *result* to the **string-concatenation** of *result* and the **substring** of *string* from *endOfLastMatch*.
16. Return *result*.

22.1.3.19 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. If *regexp* is neither **undefined** nor **null**, then
 - a. Let *searcher* be ? **GetMethod**(*regexp*, @@search).

- b. If *searcher* is not **undefined**, then
 - i. Return ? **Call**(*searcher*, *regexp*, « *O* »).
3. Let *string* be ? **ToString**(*O*).
4. Let *rx* be ? **RegExpCreate**(*regexp*, **undefined**).
5. Return ? **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.

22.1.3.20 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 index *start* and running to, but not including, index *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. Let *len* be the length of *S*.
4. Let *intStart* be ? **ToIntegerOrInfinity**(*start*).
5. If *intStart* is $-\infty$, let *from* be 0.
6. Else if *intStart* < 0, let *from* be **max**(*len* + *intStart*, 0).
7. Else, let *from* be **min**(*intStart*, *len*).
8. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ? **ToIntegerOrInfinity**(*end*).
9. If *intEnd* is $-\infty$, let *to* be 0.
10. Else if *intEnd* < 0, let *to* be **max**(*len* + *intEnd*, 0).
11. Else, let *to* be **min**(*intEnd*, *len*).
12. If *from* ≥ *to*, return the empty String.
13. Return the **substring** of *S* from *from* to *to*.

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.

22.1.3.21 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 String 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 an object, such as a RegExp, that has a **@@split** method.

When the **split** method is called, the following steps are taken:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. If *separator* is neither **undefined** nor **null**, then
 - a. Let *splitter* be ? **GetMethod**(*separator*, **@@split**).
 - b. If *splitter* is not **undefined**, then

- i. Return ? `Call(splitter, separator, « O, limit »)`.
3. Let *S* be ? `ToString(O)`.
4. Let *A* be ! `ArrayCreate(0)`.
5. Let *lengthA* be 0.
6. If *limit* is **undefined**, let *lim* be $2^{32} - 1$; else let *lim* be $\mathbb{R}(\text{? ToUint32}(\textit{limit}))$.
7. Let *R* be ? `ToString(separator)`.
8. If *lim* = 0, return *A*.
9. If *separator* is **undefined**, then
 - a. Perform ! `CreateDataPropertyOrThrow(A, "0", S)`.
 - b. Return *A*.
10. Let *s* be the length of *S*.
11. If *s* = 0, then
 - a. If *R* is not the empty String, then
 - i. Perform ! `CreateDataPropertyOrThrow(A, "0", S)`.
 - b. Return *A*.
12. Let *p* be 0.
13. Let *q* be *p*.
14. Repeat, while *q* ≠ *s*,
 - a. Let *e* be `SplitMatch(S, q, R)`.
 - b. If *e* is not-matched, set *q* to *q* + 1.
 - c. Else,
 - i. **Assert**: *e* is a non-negative integer ≤ *s*.
 - ii. If *e* = *p*, set *q* to *q* + 1.
 - iii. Else,
 1. Let *T* be the substring of *S* from *p* to *q*.
 2. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(lengthA)), T)`.
 3. Set *lengthA* to *lengthA* + 1.
 4. If *lengthA* = *lim*, return *A*.
 5. Set *p* to *e*.
 6. Set *q* to *p*.
15. Let *T* be the substring of *S* from *p* to *s*.
16. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(lengthA)), T)`.
17. Return *A*.

NOTE 1

The value of *separator* may be 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. 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 the **this** value 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 **undefined**, then the result array contains just one String, which is the **this** value (converted to a String). If *limit* is not **undefined**, then the output array is truncated so that it contains no more than *limit* elements.

NOTE 2 The **split** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

22.1.3.21.1 SplitMatch (*S*, *q*, *R*)

The abstract operation SplitMatch takes arguments *S* (a String), *q* (a non-negative integer), and *R* (a String). It returns either `not-matched` or the end index of a match. It performs the following steps when called:

1. Let *r* be the number of code units in *R*.
2. Let *s* be the number of code units in *S*.
3. If $q + r > s$, return `not-matched`.
4. If there exists an integer *i* between 0 (inclusive) and *r* (exclusive) such that the code unit at index $q + i$ within *S* is different from the code unit at index *i* within *R*, return `not-matched`.
5. Return $q + r$.

22.1.3.22 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? `RequireObjectCoercible`(**this** value).
2. Let *S* be ? `ToString`(*O*).
3. Let *isRegExp* be ? `IsRegExp`(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? `ToString`(*searchString*).
6. Let *len* be the length of *S*.
7. If *position* is **undefined**, let *pos* be 0; else let *pos* be ? `ToIntegerOrInfinity`(*position*).
8. Let *start* be the result of `clamping` *pos* between 0 and *len*.
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength* = 0, return **true**.
11. Let *end* be *start* + *searchLength*.
12. If *end* > *len*, return **false**.
13. Let *substring* be the `substring` of *S* from *start* to *end*.
14. Return ! `SameValueNonNumeric`(*substring*, *searchStr*).

NOTE 1 This method returns **true** if the sequence of code units of *searchString* converted to a String is the same as the corresponding code units of this object (converted to a String) starting at index *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 extensions 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.

22.1.3.23 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 index *start* and running to, but not including, index *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. Let *len* be the length of *S*.
4. Let *intStart* be ? **ToIntegerOrInfinity**(*start*).
5. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ? **ToIntegerOrInfinity**(*end*).
6. Let *finalStart* be the result of **clamping** *intStart* between 0 and *len*.
7. Let *finalEnd* be the result of **clamping** *intEnd* between 0 and *len*.
8. Let *from* be **min**(*finalStart*, *finalEnd*).
9. Let *to* be **max**(*finalStart*, *finalEnd*).
10. Return the **substring** of *S* from *from* to *to*.

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.

22.1.3.24 String.prototype.toLocaleLowerCase ([*reserved1* [, *reserved2*]])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the **toLocaleLowerCase** method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the **toLocaleLowerCase** method is used.

This function interprets a String value as a sequence of UTF-16 encoded 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.

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.

NOTE 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.

22.1.3.25 String.prototype.toLocaleUpperCase ([*reserved1* [, *reserved2*]])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the **toLocaleUpperCase** method as specified in the ECMA-402 specification. If an ECMAScript implementation does

not include the ECMA-402 API the following specification of the **toLocaleUpperCase** method is used.

This function interprets a String value as a sequence of UTF-16 encoded 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.

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.

NOTE 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.

22.1.3.26 String.prototype.toLowerCase ()

This function interprets a String value as a sequence of UTF-16 encoded 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. Let *sText* be ! [StringToCodePoints](#)(*S*).
4. Let *lowerText* be the result of [toLowerCase](#)(*sText*), according to the Unicode Default Case Conversion algorithm.
5. Let *L* be ! [CodePointsToString](#)(*lowerText*).
6. 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.

22.1.3.27 String.prototype.toString ()

When the **toString** method is called, the following steps are taken:

1. Return ? [thisStringValue](#)(**this** value).

NOTE For a String object, the **toString** method happens to return the same thing as the **valueOf** method.

22.1.3.28 String.prototype.toUpperCase ()

This function interprets a String value as a sequence of UTF-16 encoded code points, as described in 6.1.4.

This function behaves in exactly the same way as **String.prototype.toLowerCase**, except that the String is mapped using the toUppercase algorithm of the Unicode Default Case Conversion.

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.

22.1.3.29 String.prototype.trim ()

This function interprets a String value as a sequence of UTF-16 encoded code points, as described in 6.1.4.

The following steps are taken:

1. Let *S* be the **this** value.
2. Return ? **TrimString**(*S*, start+end).

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.

22.1.3.29.1 TrimString (*string*, *where*)

The abstract operation TrimString takes arguments *string* and *where*. It interprets *string* as a sequence of UTF-16 encoded code points, as described in 6.1.4. It performs the following steps when called:

1. Let *str* be ? **RequireObjectCoercible**(*string*).
2. Let *S* be ? **ToString**(*str*).
3. If *where* is **start**, let *T* be the String value that is a copy of *S* with leading white space removed.
4. Else if *where* is **end**, let *T* be the String value that is a copy of *S* with trailing white space removed.
5. Else,
 - a. **Assert**: *where* is **start+end**.
 - b. Let *T* be the String value that is a copy of *S* with both leading and trailing white space removed.
6. Return *T*.

The definition of white space is the union of *WhiteSpace* and *LineTerminator*. When determining whether a Unicode code point is in Unicode general category “Space_Separator” (“Zs”), code unit sequences are interpreted as UTF-16 encoded code point sequences as specified in 6.1.4.

22.1.3.30 String.prototype.trimEnd ()

This function interprets a String value as a sequence of UTF-16 encoded code points, as described in 6.1.4.

The following steps are taken:

1. Let *S* be the **this** value.
2. Return ? `TrimString(S, end)`.

NOTE The **trimEnd** 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.

22.1.3.31 `String.prototype.trimStart()`

This function interprets a String value as a sequence of UTF-16 encoded code points, as described in 6.1.4.

The following steps are taken:

1. Let *S* be the **this** value.
2. Return ? `TrimString(S, start)`.

NOTE The **trimStart** 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.

22.1.3.32 `String.prototype.valueOf()`

When the **valueOf** method is called, the following steps are taken:

1. Return ? `thisStringValue(this value)`.

22.1.3.33 `String.prototype[@@iterator]()`

When the **@@iterator** method is called it returns an Iterator object (27.1.1.2) that iterates over the code points of a String value, returning each code point as a String value. The following steps are taken:

1. Let *O* be ? `RequireObjectCoercible(this value)`.
2. Let *s* be ? `Tostring(O)`.
3. Let *closure* be a new **Abstract Closure** with no parameters that captures *s* and performs the following steps when called:
 - a. Let *position* be 0.
 - b. Let *len* be the length of *s*.
 - c. Repeat, while *position* < *len*,
 - i. Let *cp* be ! `CodePointAt(s, position)`.
 - ii. Let *nextIndex* be *position* + *cp*.[[CodeUnitCount]].
 - iii. Let *resultString* be the **substring** of *s* from *position* to *nextIndex*.
 - iv. Set *position* to *nextIndex*.
 - v. Perform ? `Yield(resultString)`.
 - d. Return **undefined**.
4. Return ! `CreateIteratorFromClosure(closure, "%StringIteratorPrototype%", %StringIteratorPrototype%)`.

The value of the "name" property of this function is "[Symbol.iterator]".

22.1.4 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.

22.1.4.1 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** }.

22.1.5 String Iterator Objects

A 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.

22.1.5.1 The `%StringIteratorPrototype%` Object

The `%StringIteratorPrototype%` object:

- has properties that are inherited by all String Iterator Objects.
- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%IteratorPrototype%`.
- has the following properties:

22.1.5.1.1 `%StringIteratorPrototype%.next()`

1. Return ? [GeneratorResume](#)(this value, empty, `"%StringIteratorPrototype%"`).

22.1.5.1.2 `%StringIteratorPrototype%` [`@@toStringTag`]

The initial value of the `@@toStringTag` property is the String value **"String Iterator"**.

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

22.2 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.

22.2.1 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*_[U, N] ::

*Disjunction*_[?U, ?N]

*Disjunction*_[U, N] ::

*Alternative*_[?U, ?N]

*Alternative*_[?U, ?N] | *Disjunction*_[?U, ?N]

*Alternative*_[U, N] ::

[empty]

*Alternative*_[?U, ?N] *Term*_[?U, ?N]

*Term*_[U, N] ::

*Assertion*_[?U, ?N]

*Atom*_[?U, ?N]

*Atom*_[?U, ?N] *Quantifier*

*Assertion*_[U, N] ::

^

\$

\ b

\ B

(? = *Disjunction*_[?U, ?N])

(? ! *Disjunction*_[?U, ?N])

(? <= *Disjunction*_[?U, ?N])

(? <! *Disjunction*_[?U, ?N])

Quantifier ::

QuantifierPrefix

QuantifierPrefix ?

QuantifierPrefix ::

*

+

?

{ *DecimalDigits*_[~Sep] }

{ *DecimalDigits*_[~Sep] , }

{ *DecimalDigits*_[~Sep] , *DecimalDigits*_[~Sep] }

*Atom*_[U, N] ::

PatternCharacter

.

`\ AtomEscape`_[?U, ?N]
`CharacterClass`_[?U]
(`GroupSpecifier`_[?U] `Disjunction`_[?U, ?N])
(? : `Disjunction`_[?U, ?N])

`SyntaxCharacter` :: **one of**
`^ $ \ . * + ? () [] { } |`

`PatternCharacter` ::
`SourceCharacter` **but not** `SyntaxCharacter`

`AtomEscape`_[U, N] ::
`DecimalEscape`
`CharacterClassEscape`_[?U]
`CharacterEscape`_[?U]
`[+N]` **k** `GroupName`_[?U]

`CharacterEscape`_[U] ::
`ControlEscape`
c `ControlLetter`
0 [lookahead \notin `DecimalDigit`]
`HexEscapeSequence`
`RegExpUnicodeEscapeSequence`_[?U]
`IdentityEscape`_[?U]

`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**

`GroupSpecifier`_[U] ::
[empty]
? `GroupName`_[?U]

`GroupName`_[U] ::
< `RegExpIdentifierName`_[?U] >

`RegExpIdentifierName`_[U] ::
`RegExpIdentifierStart`_[?U]
`RegExpIdentifierName`_[?U] `RegExpIdentifierPart`_[?U]

`RegExpIdentifierStart`_[U] ::
`UnicodeIDStart`
\$

-
`\ RegExpUnicodeEscapeSequence` _[+U]
_[~U] `UnicodeLeadSurrogate UnicodeTrailSurrogate`

`RegExpIdentifierPart` _[U] ::
`UnicodeIDContinue`
\$
`\ RegExpUnicodeEscapeSequence` _[+U]
_[~U] `UnicodeLeadSurrogate UnicodeTrailSurrogate`
<ZWNJ>
<ZWJ>

`RegExpUnicodeEscapeSequence` _[U] ::
_[+U] `u HexLeadSurrogate \u HexTrailSurrogate`
_[+U] `u HexLeadSurrogate`
_[+U] `u HexTrailSurrogate`
_[+U] `u HexNonSurrogate`
_[~U] `u Hex4Digits`
_[+U] `u{ CodePoint }`

`UnicodeLeadSurrogate` ::
any Unicode code point in the inclusive range 0xD800 to 0xDBFF

`UnicodeTrailSurrogate` ::
any Unicode code point in the inclusive range 0xDC00 to 0xDFFF

Each `\u HexTrailSurrogate` for which the choice of associated `u HexLeadSurrogate` is ambiguous shall be associated with the nearest possible `u HexLeadSurrogate` that would otherwise have no corresponding `\u HexTrailSurrogate`.

`HexLeadSurrogate` ::
`Hex4Digits` but only if the MV of `Hex4Digits` is in the inclusive range 0xD800 to 0xDBFF

`HexTrailSurrogate` ::
`Hex4Digits` but only if the MV of `Hex4Digits` is in the inclusive range 0xDC00 to 0xDFFF

`HexNonSurrogate` ::
`Hex4Digits` but only if the MV of `Hex4Digits` is not in the inclusive range 0xD800 to 0xDFFF

`IdentityEscape` _[U] ::
_[+U] `SyntaxCharacter`
_[+U] `/`
_[~U] `SourceCharacter` but not `UnicodeIDContinue`

`DecimalEscape` ::
`NonZeroDigit` `DecimalDigits` _[~Sep] `opt` `[lookahead ∉ DecimalDigit]`

`CharacterClassEscape` _[U] ::
`d`

D
S
S
W
W

[+U] p{ UnicodePropertyValueExpression }

[+U] P{ UnicodePropertyValueExpression }

UnicodePropertyValueExpression ::
UnicodePropertyName = UnicodePropertyValue
LoneUnicodePropertyNameOrValue

UnicodePropertyName ::
UnicodePropertyNameCharacters

UnicodePropertyNameCharacters ::
UnicodePropertyNameCharacter UnicodePropertyNameCharacters_{opt}

UnicodePropertyValue ::
UnicodePropertyValueCharacters

LoneUnicodePropertyNameOrValue ::
UnicodePropertyValueCharacters

UnicodePropertyValueCharacters ::
UnicodePropertyValueCharacter UnicodePropertyValueCharacters_{opt}

UnicodePropertyValueCharacter ::
UnicodePropertyNameCharacter
DecimalDigit

UnicodePropertyNameCharacter ::
ControlLetter

-

CharacterClass_[U] ::
[[lookahead ≠ ^] ClassRanges_[?U]]
[^ ClassRanges_[?U]]

ClassRanges_[U] ::
[empty]
NonemptyClassRanges_[?U]

NonemptyClassRanges_[U] ::
ClassAtom_[?U]
ClassAtom_[?U] NonemptyClassRangesNoDash_[?U]
ClassAtom_[?U] - ClassAtom_[?U] ClassRanges_[?U]

*NonemptyClassRangesNoDash*_[U] ::
 *ClassAtom*_[?U]
 *ClassAtomNoDash*_[?U] *NonemptyClassRangesNoDash*_[?U]
 *ClassAtomNoDash*_[?U] - *ClassAtom*_[?U] *ClassRanges*_[?U]

*ClassAtom*_[U] ::
 -
 *ClassAtomNoDash*_[?U]

*ClassAtomNoDash*_[U] ::
 SourceCharacter but not one of \ or] or -
 \ *ClassEscape*_[?U]

*ClassEscape*_[U] ::
 b
 [+U]-
 *CharacterClassEscape*_[?U]
 *CharacterEscape*_[?U]

NOTE A number of productions in this section are given alternative definitions in section B.1.4.

22.2.1.1 Static Semantics: Early Errors

NOTE This section is amended in B.1.4.1.

Pattern :: *Disjunction*

- It is a Syntax Error if *NcapturingParens* $\geq 2^{32} - 1$.
- It is a Syntax Error if *Pattern* contains multiple *GroupSpecifiers* whose enclosed *RegExpIdentifierNames* have the same *CapturingGroupName*.

QuantifierPrefix :: { *DecimalDigits* , *DecimalDigits* }

- It is a Syntax Error if the MV of the first *DecimalDigits* is larger than the MV of the second *DecimalDigits*.

AtomEscape :: **k** *GroupName*

- It is a Syntax Error if the enclosing *Pattern* does not contain a *GroupSpecifier* with an enclosed *RegExpIdentifierName* whose *CapturingGroupName* equals the *CapturingGroupName* of the *RegExpIdentifierName* of this production's *GroupName*.

AtomEscape :: *DecimalEscape*

- It is a Syntax Error if the *CapturingGroupNumber* of *DecimalEscape* is larger than *NcapturingParens* (22.2.2.1).

NonemptyClassRanges :: *ClassAtom* - *ClassAtom* *ClassRanges*

- It is a Syntax Error if *IsCharacterClass* of the first *ClassAtom* is **true** or *IsCharacterClass* of the second *ClassAtom*

is **true**.

- It is a Syntax Error if **IsCharacterClass** of the first *ClassAtom* is **false** and **IsCharacterClass** of the second *ClassAtom* is **false** and the **CharacterValue** of the first *ClassAtom* is larger than the **CharacterValue** of the second *ClassAtom*.

NonemptyClassRangesNoDash :: *ClassAtomNoDash* - *ClassAtom* *ClassRanges*

- It is a Syntax Error if **IsCharacterClass** of *ClassAtomNoDash* is **true** or **IsCharacterClass** of *ClassAtom* is **true**.
- It is a Syntax Error if **IsCharacterClass** of *ClassAtomNoDash* is **false** and **IsCharacterClass** of *ClassAtom* is **false** and the **CharacterValue** of *ClassAtomNoDash* is larger than the **CharacterValue** of *ClassAtom*.

*RegExpIdentifierStart*_[U] :: \ *RegExpUnicodeEscapeSequence*_[+U]

- It is a Syntax Error if the **CharacterValue** of *RegExpUnicodeEscapeSequence* is not the code point value of "\$", "_", or some code point matched by the *UnicodeIDStart* lexical grammar production.

*RegExpIdentifierStart*_[U] :: *UnicodeLeadSurrogate* *UnicodeTrailSurrogate*

- It is a Syntax Error if the result of performing **UTF16SurrogatePairToCodePoint** on the two code points matched by *UnicodeLeadSurrogate* and *UnicodeTrailSurrogate* respectively is not matched by the *UnicodeIDStart* lexical grammar production.

*RegExpIdentifierPart*_[U] :: \ *RegExpUnicodeEscapeSequence*_[+U]

- It is a Syntax Error if the **CharacterValue** of *RegExpUnicodeEscapeSequence* is not the code point value of "\$", "_", <ZWNJ>, <ZWJ>, or some code point matched by the *UnicodeIDContinue* lexical grammar production.

*RegExpIdentifierPart*_[U] :: *UnicodeLeadSurrogate* *UnicodeTrailSurrogate*

- It is a Syntax Error if the result of performing **UTF16SurrogatePairToCodePoint** on the two code points matched by *UnicodeLeadSurrogate* and *UnicodeTrailSurrogate* respectively is not matched by the *UnicodeIDContinue* lexical grammar production.

UnicodePropertyValueExpression :: *UnicodePropertyName* = *UnicodePropertyValue*

- It is a Syntax Error if the **List** of Unicode code points that is **SourceText** of *UnicodePropertyName* is not identical to a **List** of Unicode code points that is a Unicode **property name** or property alias listed in the “Property name and aliases” column of [Table 56](#).
- It is a Syntax Error if the **List** of Unicode code points that is **SourceText** of *UnicodePropertyValue* is not identical to a **List** of Unicode code points that is a value or value alias for the Unicode property or property alias given by **SourceText** of *UnicodePropertyName* listed in the “Property value and aliases” column of the corresponding tables [Table 58](#) or [Table 59](#).

UnicodePropertyValueExpression :: *LoneUnicodePropertyNameOrValue*

- It is a Syntax Error if the **List** of Unicode code points that is **SourceText** of *LoneUnicodePropertyNameOrValue* is not identical to a **List** of Unicode code points that is a Unicode general category or general category alias listed in the “Property value and aliases” column of [Table 58](#), nor a binary property or binary property alias listed in the “Property name and aliases” column of [Table 57](#).

22.2.1.2 Static Semantics: CapturingGroupNumber

NOTE This section is amended in [B.1.4.1](#).

DecimalEscape :: *NonZeroDigit*

1. Return the MV of *NonZeroDigit*.

DecimalEscape :: *NonZeroDigit* *DecimalDigits*

1. Let n be the number of code points in *DecimalDigits*.
2. Return (the MV of *NonZeroDigit* $\times 10^n$ plus the MV of *DecimalDigits*).

The definitions of “the MV of *NonZeroDigit*” and “the MV of *DecimalDigits*” are in [12.8.3](#).

22.2.1.3 Static Semantics: IsCharacterClass

NOTE This section is amended in [B.1.4.2](#).

ClassAtom :: -

ClassAtomNoDash :: *SourceCharacter* but not one of \ or] or -

ClassEscape :: **b**

ClassEscape :: -

ClassEscape :: *CharacterEscape*

1. Return **false**.

ClassEscape :: *CharacterClassEscape*

1. Return **true**.

22.2.1.4 Static Semantics: CharacterValue

NOTE 1 This section is amended in [B.1.4.3](#).

ClassAtom :: -

1. Return the code point value of U+002D (HYPHEN-MINUS).

ClassAtomNoDash :: *SourceCharacter* but not one of \ or] or -

1. Let ch be the code point matched by *SourceCharacter*.
2. Return the code point value of ch .

ClassEscape :: **b**

1. Return the code point value of U+0008 (BACKSPACE).

ClassEscape :: -

1. Return the code point value of U+002D (HYPHEN-MINUS).

CharacterEscape :: *ControlEscape*

1. Return the code point value according to [Table 55](#).

Table 55: ControlEscape Code Point Values

ControlEscape	Code Point Value	Code Point	Unicode Name	Symbol
t	9	U+0009	CHARACTER TABULATION	<HT>
n	10	U+000A	LINE FEED (LF)	<LF>
v	11	U+000B	LINE TABULATION	<VT>
f	12	U+000C	FORM FEED (FF)	<FF>
r	13	U+000D	CARRIAGE RETURN (CR)	<CR>

CharacterEscape :: **c** *ControlLetter*

1. Let *ch* be the code point matched by *ControlLetter*.
2. Let *i* be *ch*'s code point value.
3. Return the remainder of dividing *i* by 32.

CharacterEscape :: **0** [lookahead \notin *DecimalDigit*]

1. Return the code point value of U+0000 (NULL).

NOTE 2 **\0** represents the <NUL> character and cannot be followed by a decimal digit.

CharacterEscape :: *HexEscapeSequence*

1. Return the MV of *HexEscapeSequence*.

RegExpUnicodeEscapeSequence :: **u** *HexLeadSurrogate* \ **u** *HexTrailSurrogate*

1. Let *lead* be the [CharacterValue](#) of *HexLeadSurrogate*.
2. Let *trail* be the [CharacterValue](#) of *HexTrailSurrogate*.
3. Let *cp* be [UTF16SurrogatePairToCodePoint](#)(*lead*, *trail*).
4. Return the code point value of *cp*.

RegExpUnicodeEscapeSequence :: **u** *Hex4Digits*

1. Return the MV of *Hex4Digits*.

RegExpUnicodeEscapeSequence :: **u**{ *CodePoint* }

1. Return the MV of *CodePoint*.

HexLeadSurrogate :: *Hex4Digits*

HexTrailSurrogate :: *Hex4Digits*

HexNonSurrogate :: *Hex4Digits*

1. Return the MV of *HexDigits*.

CharacterEscape :: *IdentityEscape*

1. Let *ch* be the code point matched by *IdentityEscape*.
2. Return the code point value of *ch*.

22.2.1.5 Static Semantics: SourceText

UnicodePropertyNameCharacters :: *UnicodePropertyNameCharacter* *UnicodePropertyNameCharacters*_{opt}

UnicodePropertyValueCharacters :: *UnicodePropertyValueCharacter* *UnicodePropertyValueCharacters*_{opt}

1. Return the [List](#), in source text order, of Unicode code points in the source text matched by this production.

22.2.1.6 Static Semantics: CapturingGroupName

*RegExpIdentifierName*_[U] ::

*RegExpIdentifierStart*_[?U]

*RegExpIdentifierName*_[?U] *RegExpIdentifierPart*_[?U]

1. Let *idText* be the source text matched by *RegExpIdentifierName*.
2. Let *idTextUnescaped* be the result of replacing any occurrences of \backslash *RegExpUnicodeEscapeSequence* in *idText* with the code point represented by the *RegExpUnicodeEscapeSequence*.
3. Return ! [CodePointsToString](#)(*idTextUnescaped*).

22.2.2 Pattern Semantics

NOTE 1 This section is amended in [B.1.4.4](#).

A regular expression pattern is converted into an [Abstract Closure](#) 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 [Abstract Closure](#) is used as the value of a `RegExp` object's `[[RegExpMatcher]]` internal slot.

A *Pattern* is either a BMP pattern or a Unicode pattern depending upon whether or not its associated flags contain a **u**. A BMP pattern matches against a `String` interpreted as consisting of a sequence of 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 ([6.1.4](#)). In either context, “character value” means the numeric value of the corresponding non-encoded code point.

The syntax and semantics of *Pattern* is defined as if the source code for the *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 2

For example, consider a pattern expressed in source text 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 [UTF16SurrogatePairToCodePoint](#) must be used in producing a [List](#) whose sole element is 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.

22.2.2.1 Notation

The descriptions below use the following aliases:

- *Input* is a [List](#) whose elements are the characters 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 *Atom* :: (*GroupSpecifier Disjunction*) Parse Nodes) in the pattern. A left-capturing parenthesis is any **(** pattern character that is matched by the **(** terminal of the *Atom* :: (*GroupSpecifier Disjunction*) production.
- *DotAll* is **true** if the RegExp object's `[[OriginalFlags]]` internal slot contains "s" and otherwise is **false**.
- *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**.
- *WordCharacters* is the mathematical set that is the union of all sixty-three characters in "ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_" (letters, numbers, and U+005F (LOW LINE) in the Unicode Basic Latin block) and all characters *c* for which *c* is not in that set but [Canonicalize](#)(*c*) is. *WordCharacters* cannot contain more than sixty-three characters unless *Unicode* and *IgnoreCase* are both **true**.

Furthermore, the descriptions below use the following internal data structures:

- A *CharSet* is a mathematical set of characters. When the *Unicode* flag is **true**, "all characters" means the CharSet containing all code point values; otherwise "all characters" means the CharSet containing all code unit values.
- 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](#) of characters 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* is an [Abstract Closure](#) that takes one State argument and returns a *MatchResult* result. The Continuation attempts to match the remaining portion (specified by the closure's captured values) 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* is an [Abstract 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 captured values) 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.

22.2.2.2 Pattern

The production *Pattern* :: *Disjunction* evaluates as follows:

1. Evaluate *Disjunction* with 1 as its *direction* argument to obtain a Matcher *m*.
2. Return a new [Abstract Closure](#) with parameters (*str*, *index*) that captures *m* and performs the following steps when called:
 - a. **Assert:** *Type(str)* is String.
 - b. **Assert:** *index* is a non-negative *integer* which is \leq the length of *str*.
 - c. If *Unicode* is **true**, let *Input* be ! [StringToCodePoints\(str\)](#). Otherwise, let *Input* be a [List](#) whose elements are the code units that are the elements of *str*. *Input* will be used throughout the algorithms in 22.2.2. Each element of *Input* is considered to be a character.
 - d. Let *InputLength* be the number of characters contained in *Input*. This alias will be used throughout the algorithms in 22.2.2.
 - e. Let *listIndex* be the index into *Input* of the character that was obtained from element *index* of *str*.
 - f. Let *c* be a new Continuation with parameters (*y*) that captures nothing and performs the following steps when called:
 - i. **Assert:** *y* is a State.
 - ii. Return *y*.
 - g. Let *cap* be a [List](#) of *NcapturingParens* **undefined** values, indexed 1 through *NcapturingParens*.
 - h. Let *x* be the State (*listIndex*, *cap*).
 - i. Return *m(x, c)*.

NOTE

A Pattern evaluates (“compiles”) to an [Abstract Closure](#) value. [RegExpBuiltinExec](#) 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 22.2.2 are designed so that compiling a pattern may throw a **SyntaxError** exception; on the other hand, once the pattern is successfully compiled, applying the resulting [Abstract Closure](#) to find a match in a String cannot throw an exception (except for any [implementation-defined](#) exceptions that can occur anywhere such as out-of-memory).

22.2.2.3 Disjunction

With parameter *direction*.

The production *Disjunction* :: *Alternative* evaluates as follows:

1. Evaluate *Alternative* with argument *direction* to obtain a Matcher *m*.
2. Return *m*.

The production *Disjunction* :: *Alternative* | *Disjunction* evaluates as follows:

1. Evaluate *Alternative* with argument *direction* to obtain a Matcher *m1*.
2. Evaluate *Disjunction* with argument *direction* to obtain a Matcher *m2*.
3. Return a new Matcher with parameters (*x*, *c*) that captures *m1* and *m2* and performs the following steps when called:
 - a. **Assert:** *x* is a State.
 - b. **Assert:** *c* is a Continuation.
 - c. Let *r* be *m1*(*x*, *c*).
 - d. If *r* is not **failure**, return *r*.
 - e. Return *m2*(*x*, *c*).

NOTE

The **|** regular expression operator separates two alternatives. The pattern first tries to match the left *Alternative* (followed by the sequel of the regular expression); if it fails, it tries to match the right *Disjunction* (followed by the sequel of the regular expression). If the left *Alternative*, the right *Disjunction*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*. Any capturing parentheses inside a portion of the pattern skipped by **|** produce **undefined** values instead of Strings. Thus, for example,

```
/a|ab/.exec("abc")
```

returns the result "a" and not "ab". Moreover,

```
/((a)|(ab))((c)|(bc))/.exec("abc")
```

returns the array

```
["abc", "a", "a", undefined, "bc", undefined, "bc"]
```

and not

```
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

The order in which the two alternatives are tried is independent of the value of *direction*.

22.2.2.4 Alternative

With parameter *direction*.

The production *Alternative* :: [empty] evaluates as follows:

1. Return a new Matcher with parameters (*x*, *c*) that captures nothing and performs the following steps when

called:

- a. **Assert:** x is a State.
- b. **Assert:** c is a Continuation.
- c. Return $c(x)$.

The production $Alternative :: Alternative Term$ evaluates as follows:

1. Evaluate $Alternative$ with argument $direction$ to obtain a Matcher $m1$.
2. Evaluate $Term$ with argument $direction$ to obtain a Matcher $m2$.
3. If $direction = 1$, then
 - a. Return a new Matcher with parameters (x, c) that captures $m1$ and $m2$ and performs the following steps when called:
 - i. **Assert:** x is a State.
 - ii. **Assert:** c is a Continuation.
 - iii. Let d be a new Continuation with parameters (y) that captures c and $m2$ and performs the following steps when called:
 1. **Assert:** y is a State.
 2. Return $m2(y, c)$.
 - iv. Return $m1(x, d)$.
4. Else,
 - a. **Assert:** $direction$ is -1.
 - b. Return a new Matcher with parameters (x, c) that captures $m1$ and $m2$ and performs the following steps when called:
 - i. **Assert:** x is a State.
 - ii. **Assert:** c is a Continuation.
 - iii. Let d be a new Continuation with parameters (y) that captures c and $m1$ and performs the following steps when called:
 1. **Assert:** y is a State.
 2. Return $m1(y, c)$.
 - iv. Return $m2(x, d)$.

NOTE

Consecutive *Terms* try to simultaneously match consecutive portions of *Input*. When $direction = 1$, 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*. When $direction = -1$, the evaluation order of *Alternative* and *Term* are reversed.

22.2.2.5 Term

With parameter $direction$.

The production $Term :: Assertion$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $Assertion$.

NOTE

The resulting Matcher is independent of $direction$.

The production $Term :: Atom$ evaluates as follows:

1. Return the Matcher that is the result of evaluating *Atom* with argument *direction*.

The production *Term* :: *Atom Quantifier* evaluates as follows:

1. Evaluate *Atom* with argument *direction* to obtain a Matcher *m*.
2. Evaluate *Quantifier* to obtain the three results: a non-negative integer *min*, a non-negative integer (or $+\infty$) *max*, and Boolean *greedy*.
3. Assert: $min \leq max$.
4. Let *parenIndex* be the number of left-capturing parentheses in the entire regular expression that occur to the left of this *Term*. This is the total number of *Atom* :: (*GroupSpecifier Disjunction*) Parse Nodes prior to or enclosing this *Term*.
5. Let *parenCount* be the number of left-capturing parentheses in *Atom*. This is the total number of *Atom* :: (*GroupSpecifier Disjunction*) Parse Nodes enclosed by *Atom*.
6. Return a new Matcher with parameters (*x*, *c*) that captures *m*, *min*, *max*, *greedy*, *parenIndex*, and *parenCount* and performs the following steps when called:
 - a. Assert: *x* is a State.
 - b. Assert: *c* is a Continuation.
 - c. Return ! RepeatMatcher(*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*).

22.2.2.5.1 RepeatMatcher (*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*)

The abstract operation RepeatMatcher takes arguments *m* (a Matcher), *min* (a non-negative integer), *max* (a non-negative integer or $+\infty$), *greedy* (a Boolean), *x* (a State), *c* (a Continuation), *parenIndex* (a non-negative integer), and *parenCount* (a non-negative integer). It performs the following steps when called:

1. If $max = 0$, return $c(x)$.
2. Let *d* be a new Continuation with parameters (*y*) that captures *m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, and *parenCount* and performs the following steps when called:
 - a. Assert: *y* is a State.
 - b. If $min = 0$ and *y*'s *endIndex* = *x*'s *endIndex*, return failure.
 - c. If $min = 0$, let *min2* be 0; otherwise let *min2* be *min* - 1.
 - d. If *max* is $+\infty$, let *max2* be $+\infty$; otherwise let *max2* be *max* - 1.
 - e. Return ! RepeatMatcher(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*).
3. Let *cap* be a copy of *x*'s captures List.
4. For each integer *k* such that $parenIndex < k$ and $k \leq parenIndex + parenCount$, set *cap*[*k*] to **undefined**.
5. Let *e* be *x*'s *endIndex*.
6. Let *xr* be the State (*e*, *cap*).
7. If $min \neq 0$, return $m(xr, d)$.
8. If *greedy* is **false**, then
 - a. Let *z* be $c(x)$.
 - b. If *z* is not failure, return *z*.
 - c. Return $m(xr, d)$.
9. Let *z* be $m(xr, d)$.
10. If *z* is not failure, return *z*.
11. Return $c(x)$.

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 ($(n - 1)^{\text{st}}$) repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the $(n - 1)^{\text{st}}$ repetition of *Atom* and so on.

Compare

```
/a[a-z]{2,4}/.exec("abcdefghi")
```

which returns **"abcde"** with

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

which returns **"abc"**.

Consider also

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

which, by the choice point ordering above, returns the array

```
["aaba", "ba"]
```

and not any of:

```
["aabaac", "aabaac"]
```

```
["aabaac", "c"]
```

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaa,aaaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/, "$1")
```

which returns the gcd in unary notation **"aaaaa"**.

NOTE 3 Step 4 of the RepeatMatcher clears *Atom*'s captures each time *Atom* is repeated. We can see its behaviour in the regular expression

```
/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

which returns the array

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost ***** clears all captured Strings contained in the quantified *Atom*, which in this case includes capture Strings numbered 2, 3, 4, and 5.

NOTE 4 Step 2.b of the RepeatMatcher 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\1+/.exec("baaaac")
```

which returns the array

```
["b", ""]
```

22.2.2.6 Assertion

The production *Assertion* :: **^** evaluates as follows:

1. Return a new Matcher with parameters (*x*, *c*) that captures nothing and performs the following steps when called:
 - a. **Assert**: *x* is a State.
 - b. **Assert**: *c* is a Continuation.
 - c. Let *e* be *x*'s *endIndex*.
 - d. If *e* = 0, or if *Multiline* is **true** and the character *Input*[*e* - 1] is one of *LineTerminator*, then
 - i. Return *c*(*x*).
 - e. Return failure.

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 as follows:

1. Return a new Matcher with parameters (x, c) that captures nothing and performs the following steps when called:
 - a. **Assert:** x is a State.
 - b. **Assert:** c is a Continuation.
 - c. Let e be x 's *endIndex*.
 - d. If $e = \text{InputLength}$, or if *Multiline* is **true** and the character $\text{Input}[e]$ is one of *LineTerminator*, then
 - i. Return $c(x)$.
 - e. Return **failure**.

The production *Assertion* :: \ **b** evaluates as follows:

1. Return a new Matcher with parameters (x, c) that captures nothing and performs the following steps when called:
 - a. **Assert:** x is a State.
 - b. **Assert:** c is a Continuation.
 - c. Let e be x 's *endIndex*.
 - d. Let a be ! *IsWordChar*($e - 1$).
 - e. Let b be ! *IsWordChar*(e).
 - f. If a is **true** and b is **false**, or if a is **false** and b is **true**, return $c(x)$.
 - g. Return **failure**.

The production *Assertion* :: \ **B** evaluates as follows:

1. Return a new Matcher with parameters (x, c) that captures nothing and performs the following steps when called:
 - a. **Assert:** x is a State.
 - b. **Assert:** c is a Continuation.
 - c. Let e be x 's *endIndex*.
 - d. Let a be ! *IsWordChar*($e - 1$).
 - e. Let b be ! *IsWordChar*(e).
 - f. If a is **true** and b is **true**, or if a is **false** and b is **false**, return $c(x)$.
 - g. Return **failure**.

The production *Assertion* :: (? = *Disjunction*) evaluates as follows:

1. Evaluate *Disjunction* with 1 as its *direction* argument to obtain a Matcher m .
2. Return a new Matcher with parameters (x, c) that captures m and performs the following steps when called:
 - a. **Assert:** x is a State.
 - b. **Assert:** c is a Continuation.
 - c. Let d be a new Continuation with parameters (y) that captures nothing and performs the following steps when called:
 - i. **Assert:** y is a State.
 - ii. Return y .
 - d. Let r be $m(x, d)$.
 - e. If r is **failure**, return **failure**.
 - f. Let y be r 's State.
 - g. Let cap be y 's *captures List*.
 - h. Let xe be x 's *endIndex*.
 - i. Let z be the State (xe, cap).
 - j. Return $c(z)$.

The production *Assertion* :: (? ! *Disjunction*) evaluates as follows:

1. Evaluate *Disjunction* with 1 as its *direction* argument to obtain a Matcher *m*.
2. Return a new Matcher with parameters (*x*, *c*) that captures *m* and performs the following steps when called:
 - a. **Assert:** *x* is a State.
 - b. **Assert:** *c* is a Continuation.
 - c. Let *d* be a new Continuation with parameters (*y*) that captures nothing and performs the following steps when called:
 - i. **Assert:** *y* is a State.
 - ii. Return *y*.
 - d. Let *r* be *m(x, d)*.
 - e. If *r* is not failure, return failure.
 - f. Return *c(x)*.

The production *Assertion* :: (? <= *Disjunction*) evaluates as follows:

1. Evaluate *Disjunction* with -1 as its *direction* argument to obtain a Matcher *m*.
2. Return a new Matcher with parameters (*x*, *c*) that captures *m* and performs the following steps when called:
 - a. **Assert:** *x* is a State.
 - b. **Assert:** *c* is a Continuation.
 - c. Let *d* be a new Continuation with parameters (*y*) that captures nothing and performs the following steps when called:
 - i. **Assert:** *y* is a State.
 - ii. Return *y*.
 - d. Let *r* be *m(x, d)*.
 - e. If *r* is failure, return failure.
 - f. Let *y* be *r*'s State.
 - g. Let *cap* be *y*'s *captures List*.
 - h. Let *xe* be *x*'s *endIndex*.
 - i. Let *z* be the State (*xe, cap*).
 - j. Return *c(z)*.

The production *Assertion* :: (? <! *Disjunction*) evaluates as follows:

1. Evaluate *Disjunction* with -1 as its *direction* argument to obtain a Matcher *m*.
2. Return a new Matcher with parameters (*x*, *c*) that captures *m* and performs the following steps when called:
 - a. **Assert:** *x* is a State.
 - b. **Assert:** *c* is a Continuation.
 - c. Let *d* be a new Continuation with parameters (*y*) that captures nothing and performs the following steps when called:
 - i. **Assert:** *y* is a State.
 - ii. Return *y*.
 - d. Let *r* be *m(x, d)*.
 - e. If *r* is not failure, return failure.
 - f. Return *c(x)*.

22.2.2.6.1 IsWordChar (*e*)

The abstract operation IsWordChar takes argument *e* (an *integer*). It performs the following steps when called:

1. If $e = -1$ or e is *InputLength*, return **false**.
2. Let c be the character *Input*[e].
3. If c is in *WordCharacters*, return **true**.
4. Return **false**.

22.2.2.7 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 12.8.3).
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 .

22.2.2.8 Atom

With parameter *direction*.

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. Return ! **CharacterSetMatcher**(A , **false**, *direction*).

The production $Atom :: \cdot$ evaluates as follows:

1. Let A be the CharSet of all characters.
2. If $DotAll$ is not **true**, then
 - a. Remove from A all characters corresponding to a code point on the right-hand side of the *LineTerminator* production.
3. Return ! `CharacterSetMatcher`(A , **false**, $direction$).

The production $Atom :: \backslash AtomEscape$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $AtomEscape$ with argument $direction$.

The production $Atom :: CharacterClass$ evaluates as follows:

1. Evaluate $CharacterClass$ to obtain a CharSet A and a Boolean $invert$.
2. Return ! `CharacterSetMatcher`(A , $invert$, $direction$).

The production $Atom :: (GroupSpecifier Disjunction)$ evaluates as follows:

1. Evaluate $Disjunction$ with argument $direction$ 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 $Atom$. This is the total number of $Atom :: (GroupSpecifier Disjunction)$ Parse Nodes prior to or enclosing this $Atom$.
3. Return a new Matcher with parameters (x, c) that captures $direction$, m , and $parenIndex$ and performs the following steps when called:
 - a. **Assert:** x is a State.
 - b. **Assert:** c is a Continuation.
 - c. Let d be a new Continuation with parameters (y) that captures x , c , $direction$, and $parenIndex$ and performs the following steps when called:
 - i. **Assert:** y is a State.
 - ii. Let cap be a copy of y 's *captures List*.
 - iii. Let xe be x 's *endIndex*.
 - iv. Let ye be y 's *endIndex*.
 - v. If $direction = 1$, then
 1. **Assert:** $xe \leq ye$.
 2. Let s be a *List* whose elements are the characters of *Input* at indices xe (inclusive) through ye (exclusive).
 - vi. Else,
 1. **Assert:** $direction$ is -1.
 2. **Assert:** $ye \leq xe$.
 3. Let s be a *List* whose elements are the characters of *Input* at indices ye (inclusive) through xe (exclusive).
 - vii. Set $cap[parenIndex + 1]$ to s .
 - viii. Let z be the State (ye, cap) .
 - ix. Return $c(z)$.
 - d. Return $m(x, d)$.

The production $Atom :: (? : Disjunction)$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $Disjunction$ with argument $direction$.

22.2.2.8.1 CharacterSetMatcher (*A*, *invert*, *direction*)

The abstract operation CharacterSetMatcher takes arguments *A* (a CharSet), *invert* (a Boolean), and *direction* (1 or -1). It performs the following steps when called:

1. Return a new Matcher with parameters (*x*, *c*) that captures *A*, *invert*, and *direction* and performs the following steps when called:
 - a. **Assert**: *x* is a State.
 - b. **Assert**: *c* is a Continuation.
 - c. Let *e* be *x*'s *endIndex*.
 - d. Let *f* be *e* + *direction*.
 - e. If *f* < 0 or *f* > *InputLength*, return failure.
 - f. Let *index* be *min*(*e*, *f*).
 - g. Let *ch* be the character *Input*[*index*].
 - h. Let *cc* be *Canonicalize*(*ch*).
 - i. If there exists a member *a* of *A* such that *Canonicalize*(*a*) is *cc*, let *found* be **true**. Otherwise, let *found* be **false**.
 - j. If *invert* is **false** and *found* is **false**, return failure.
 - k. If *invert* is **true** and *found* is **true**, return failure.
 - l. Let *cap* be *x*'s *captures List*.
 - m. Let *y* be the State (*f*, *cap*).
 - n. Return *c*(*y*).

22.2.2.8.2 Canonicalize (*ch*)

The abstract operation Canonicalize takes argument *ch* (a character). It performs the following steps when called:

1. If *Unicode* is **true** and *IgnoreCase* is **true**, then
 - a. If the file CaseFolding.txt of the Unicode Character Database provides a simple or common case folding mapping for *ch*, return the result of applying that mapping to *ch*.
 - b. Return *ch*.
2. If *IgnoreCase* is **false**, return *ch*.
3. **Assert**: *ch* is a UTF-16 code unit.
4. Let *cp* be the code point whose numeric value is that of *ch*.
5. Let *u* be the result of *toUpperCase*(« *cp* »), according to the Unicode Default Case Conversion algorithm.
6. Let *uStr* be ! *CodePointsToString*(*u*).
7. If *uStr* does not consist of a single code unit, return *ch*.
8. Let *cu* be *uStr*'s single code unit element.
9. If the numeric value of *ch* ≥ 128 and the numeric value of *cu* < 128, return *ch*.
10. 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 non-zero decimal number), referenced in a replace String, or returned as part of an array from the regular expression matching **Abstract Closure**. To inhibit the capturing behaviour of parentheses, use the form **(? : *Disjunction*)** instead.

NOTE 2

The form **(?= Disjunction)** specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside *Disjunction* must match at the current position, but the current position is not advanced before matching the sequel. If *Disjunction* can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a **(?=** form (this unusual behaviour is inherited from Perl). This only matters when the *Disjunction* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

```
/(?=(a+))/ .exec("baaabac")
```

matches the empty String immediately after the first **b** and therefore returns the array:

```
["", "aaa"]
```

To illustrate the lack of backtracking into the lookahead, consider:

```
/(?=(a+))a*b\1/ .exec("baaabac")
```

This expression returns

```
["aba", "a"]
```

and not:

```
["aaaba", "a"]
```

NOTE 3

The form **(?! Disjunction)** specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside *Disjunction* must fail to match at the current position. The current position is not advanced before matching the sequel. *Disjunction* can contain capturing parentheses, but backreferences to them only make sense from within *Disjunction* itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **undefined** because the negative lookahead must fail for the pattern to succeed. For example,

```
/(.*?)a(?!(a+)b\2c)\2(.*)/ .exec("baaabaac")
```

looks for an **a** not immediately followed by some positive number *n* of **a**'s, a **b**, another *n* **a**'s (specified by the first **\2**) and a **c**. The second **\2** is outside the negative lookahead, so it matches against **undefined** and therefore always succeeds. The whole expression returns the array:

```
["baaabaac", "ba", undefined, "abaac"]
```

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+00DF) to **SS**. It may however map a code point outside the Basic Latin range to a character within, for example, **f** (U+017F) 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]/ui`.

22.2.2.8.3 UnicodeMatchProperty (*p*)

The abstract operation UnicodeMatchProperty takes argument *p* (a List of Unicode code points). It performs the following steps when called:

1. **Assert:** *p* is a List of Unicode code points that is identical to a List of Unicode code points that is a Unicode property name or property alias listed in the “Property name and aliases” column of Table 56 or Table 57.
2. Let *c* be the canonical property name of *p* as given in the “Canonical property name” column of the corresponding row.
3. Return the List of Unicode code points of *c*.

Implementations must support the Unicode property names and aliases listed in Table 56 and Table 57. To ensure interoperability, implementations must not support any other property names or aliases.

NOTE 1 For example, **Script_Extensions** (property name) and **scx** (property alias) are valid, but **script_extensions** or **Scx** aren't.

NOTE 2 The listed properties form a superset of what UTS18 RL1.2 requires.

Table 56: Non-binary Unicode property aliases and their canonical property names

Property name and aliases	Canonical property name
General_Category	General_Category
gc	
Script	Script
sc	
Script_Extensions	Script_Extensions
scx	

Table 57: Binary Unicode property aliases and their canonical property names

Property name and aliases	Canonical property name
ASCII	ASCII
ASCII_Hex_Digit	ASCII_Hex_Digit
AHex	
Alphabetic	Alphabetic

Alpha	
Any	Any
Assigned	Assigned
Bidi_Control	Bidi_Control
Bidi_C	
Bidi_Mirrored	Bidi_Mirrored
Bidi_M	
Case_Ignorable	Case_Ignorable
CI	
Cased	Cased
Changes_When_Casefolded	Changes_When_Casefolded
CWCF	
Changes_When_Casemapped	Changes_When_Casemapped
CWCM	
Changes_When_Lowercased	Changes_When_Lowercased
CWL	
Changes_When_NFKC_Casefolded	Changes_When_NFKC_Casefolded
CWKCF	
Changes_When_Titlecased	Changes_When_Titlecased
CWT	
Changes_When_Uppercased	Changes_When_Uppercased
CWU	
Dash	Dash
Default_Ignorable_Code_Point	Default_Ignorable_Code_Point
DI	
Deprecated	Deprecated
Dep	
Diacritic	Diacritic
Dia	
Emoji	Emoji
Emoji_Component	Emoji_Component
EComp	
Emoji_Modifier	Emoji_Modifier
EMod	
Emoji_Modifier_Base	Emoji_Modifier_Base
EBase	

Emoji_Presentation	Emoji_Presentation
EPres	
Extended_Pictographic	Extended_Pictographic
ExtPict	
Extender	Extender
Ext	
Grapheme_Base	Grapheme_Base
Gr_Base	
Grapheme_Extend	Grapheme_Extend
Gr_Ext	
Hex_Digit	Hex_Digit
Hex	
IDS_Binary_Operator	IDS_Binary_Operator
IDSB	
IDS_Tertiary_Operator	IDS_Tertiary_Operator
IDST	
ID_Continue	ID_Continue
IDC	
ID_Start	ID_Start
IDS	
Ideographic	Ideographic
Ideo	
Join_Control	Join_Control
Join_C	
Logical_Order_Exception	Logical_Order_Exception
LOE	
Lowercase	Lowercase
Lower	
Math	Math
Noncharacter_Code_Point	Noncharacter_Code_Point
NChar	
Pattern_Syntax	Pattern_Syntax
Pat_Syn	
Pattern_White_Space	Pattern_White_Space
Pat_WS	
Quotation_Mark	Quotation_Mark

QMark	
Radical	Radical
Regional_Indicator	Regional_Indicator
RI	
Sentence_Terminal	Sentence_Terminal
STerm	
Soft_Dotted	Soft_Dotted
SD	
Terminal_Punctuation	Terminal_Punctuation
Term	
Unified_Ideograph	Unified_Ideograph
UIdeo	
Uppercase	Uppercase
Upper	
Variation_Selector	Variation_Selector
VS	
White_Space	White_Space
space	
XID_Continue	XID_Continue
XIDC	
XID_Start	XID_Start
XIDS	

22.2.2.8.4 UnicodeMatchPropertyValue (*p*, *v*)

The abstract operation UnicodeMatchPropertyValue takes arguments *p* (a List of Unicode code points) and *v* (a List of Unicode code points). It performs the following steps when called:

1. **Assert:** *p* is a List of Unicode code points that is identical to a List of Unicode code points that is a canonical, unaliased Unicode property name listed in the “Canonical property name” column of Table 56.
2. **Assert:** *v* is a List of Unicode code points that is identical to a List of Unicode code points that is a property value or property value alias for Unicode property *p* listed in the “Property value and aliases” column of Table 58 or Table 59.
3. Let *value* be the canonical property value of *v* as given in the “Canonical property value” column of the corresponding row.
4. Return the List of Unicode code points of *value*.

Implementations must support the Unicode property value names and aliases listed in Table 58 and Table 59. To ensure interoperability, implementations must not support any other property value names or aliases.

NOTE 1 For example, **Xpeo** and **Old_Persian** are valid **Script_Extensions** values, but **xpeo** and **Old Persian** aren't.

NOTE 2 This algorithm differs from [the matching rules for symbolic values listed in UAX44](#): case, white space, U+002D (HYPHEN-MINUS), and U+005F (LOW LINE) are not ignored, and the **Is** prefix is not supported.

Table 58: Value aliases and canonical values for the Unicode property **General_Category**

Property value and aliases	Canonical property value
Cased_Letter	Cased_Letter
LC	
Close_Punctuation	Close_Punctuation
Pe	
Connector_Punctuation	Connector_Punctuation
Pc	
Control	Control
Cc	
cntrl	
Currency_Symbol	Currency_Symbol
Sc	
Dash_Punctuation	Dash_Punctuation
Pd	
Decimal_Number	Decimal_Number
Nd	
digit	
Enclosing_Mark	Enclosing_Mark
Me	
Final_Punctuation	Final_Punctuation
Pf	
Format	Format
Cf	
Initial_Punctuation	Initial_Punctuation
Pi	
Letter	Letter
L	
Letter_Number	Letter_Number
NL	

Line_Separator	Line_Separator
Zl	
Lowercase_Letter	Lowercase_Letter
Ll	
Mark	Mark
M	
Combining_Mark	
Math_Symbol	Math_Symbol
Sm	
Modifier_Letter	Modifier_Letter
Lm	
Modifier_Symbol	Modifier_Symbol
Sk	
Nonspacing_Mark	Nonspacing_Mark
Mn	
Number	Number
N	
Open_Punctuation	Open_Punctuation
Ps	
Other	Other
C	
Other_Letter	Other_Letter
Lo	
Other_Number	Other_Number
No	
Other_Punctuation	Other_Punctuation
Po	
Other_Symbol	Other_Symbol
So	
Paragraph_Separator	Paragraph_Separator
Zp	
Private_Use	Private_Use
Co	
Punctuation	Punctuation
P	
punct	

Separator	Separator
Z	
Space_Separator	Space_Separator
Zs	
Spacing_Mark	Spacing_Mark
Mc	
Surrogate	Surrogate
Cs	
Symbol	Symbol
S	
Titlecase_Letter	Titlecase_Letter
Lt	
Unassigned	Unassigned
Cn	
Uppercase_Letter	Uppercase_Letter
Lu	

Table 59: Value aliases and canonical values for the Unicode properties **Script** and **Script_Extensions**

Property value and aliases	Canonical property value
Adlam	Adlam
Adlm	
Ahom	Ahom
Anatolian_Hieroglyphs	Anatolian_Hieroglyphs
Hluw	
Arabic	Arabic
Arab	
Armenian	Armenian
Armn	
Avestan	Avestan
Avst	
Balinese	Balinese
Bali	
Bamum	Bamum
Bamu	
Bassa_Vah	Bassa_Vah
Bass	

Batak	Batak
Batk	
Bengali	Bengali
Beng	
Bhaiksuki	Bhaiksuki
Bhks	
Bopomofo	Bopomofo
Bopo	
Brahmi	Brahmi
Brah	
Braille	Braille
Brai	
Buginese	Buginese
Bugi	
Buhid	Buhid
Buhd	
Canadian_Aboriginal	Canadian_Aboriginal
Cans	
Carian	Carian
Cari	
Caucasian_Albanian	Caucasian_Albanian
Aghb	
Chakma	Chakma
Cakm	
Cham	Cham
Chorasmian	Chorasmian
Chrs	
Cherokee	Cherokee
Cher	
Common	Common
Zyyy	
Coptic	Coptic
Copt	
Qaac	
Cuneiform	Cuneiform
Xsux	

Cypriot	Cypriot
Cprt	
Cyrillic	Cyrillic
Cyrl	
Deseret	Deseret
Dsrt	
Devanagari	Devanagari
Deva	
Dives_Akuru	Dives_Akuru
Diak	
Dogra	Dogra
Dogr	
Duployan	Duployan
Dupl	
Egyptian_Hieroglyphs	Egyptian_Hieroglyphs
Egyp	
Elbasan	Elbasan
Elba	
Elymaic	Elymaic
Elym	
Ethiopic	Ethiopic
Ethi	
Georgian	Georgian
Geor	
Glagolitic	Glagolitic
Glag	
Gothic	Gothic
Goth	
Grantha	Grantha
Gran	
Greek	Greek
GreK	
Gujarati	Gujarati
Gujr	
Gunjala_Gondi	Gunjala_Gondi
Gong	

Gurmukhi	Gurmukhi
Guru	
Han	Han
Hani	
Hangul	Hangul
Hang	
Hanifi_Rohingya	Hanifi_Rohingya
Rohg	
Hanunoo	Hanunoo
Hano	
Hatran	Hatran
Hatr	
Hebrew	Hebrew
Hebr	
Hiragana	Hiragana
Hira	
Imperial_Aramaic	Imperial_Aramaic
Armi	
Inherited	Inherited
Zinh	
Qaai	
Inscriptional_Pahlavi	Inscriptional_Pahlavi
Phli	
Inscriptional_Parthian	Inscriptional_Parthian
Prti	
Javanese	Javanese
Java	
Kaithi	Kaithi
Kthi	
Kannada	Kannada
Knda	
Katakana	Katakana
Kana	
Kayah_Li	Kayah_Li
Kali	
Kharoshthi	Kharoshthi

Khar	
Khitan_Small_Script	Khitan_Small_Script
Kits	
Khmer	Khmer
Khmr	
Khojki	Khojki
Khoj	
Khudawadi	Khudawadi
Sind	
Lao	Lao
Laoo	
Latin	Latin
Latn	
Lepcha	Lepcha
Lepc	
Limbu	Limbu
Limb	
Linear_A	Linear_A
Lina	
Linear_B	Linear_B
Linb	
Lisu	Lisu
Lycian	Lycian
Lyci	
Lydian	Lydian
Lydi	
Mahajani	Mahajani
Mahj	
Makasar	Makasar
Maka	
Malayalam	Malayalam
Mlym	
Mandaic	Mandaic
Mand	
Manichaean	Manichaean
Mani	

Marchen	Marchen
Marc	
Medefaidrin	Medefaidrin
Medf	
Masaram_Gondi	Masaram_Gondi
Gonm	
Meetei_Mayek	Meetei_Mayek
Mtei	
Mende_Kikakui	Mende_Kikakui
Mend	
Meroitic_Cursive	Meroitic_Cursive
Merc	
Meroitic_Hieroglyphs	Meroitic_Hieroglyphs
Mero	
Miao	Miao
Plrd	
Modi	Modi
Mongolian	Mongolian
Mong	
Mro	Mro
Mroo	
Multani	Multani
Mult	
Myanmar	Myanmar
Mymr	
Nabataean	Nabataean
Nbat	
Nandinagari	Nandinagari
Nand	
New_Tai_Lue	New_Tai_Lue
Talu	
Newa	Newa
Nko	Nko
Nkoo	
Nushu	Nushu
Nshu	

Nyiakeng_Puachue_Hmong	Nyiakeng_Puachue_Hmong
Hmnp	
Ogham	Ogham
Ogam	
Ol_Chiki	Ol_Chiki
Olck	
Old_Hungarian	Old_Hungarian
Hung	
Old_Italic	Old_Italic
Ital	
Old_North_Arabian	Old_North_Arabian
Narb	
Old_Permic	Old_Permic
Perm	
Old_Persian	Old_Persian
Xpeo	
Old_Sogdian	Old_Sogdian
Sogo	
Old_South_Arabian	Old_South_Arabian
Sarb	
Old_Turkic	Old_Turkic
Orkh	
Oriya	Oriya
Orya	
Osage	Osage
Osge	
Osmanya	Osmanya
Osma	
Pahawh_Hmong	Pahawh_Hmong
Hmng	
Palmyrene	Palmyrene
Palm	
Pau_Cin_Hau	Pau_Cin_Hau
Pauc	
Phags_Pa	Phags_Pa
Phag	

Phoenician	Phoenician
Phnx	
Psalter_Pahlavi	Psalter_Pahlavi
Phlp	
Rejang	Rejang
Rjng	
Runic	Runic
Runr	
Samaritan	Samaritan
Samr	
Saurashtra	Saurashtra
Saur	
Sharada	Sharada
Shrd	
Shavian	Shavian
Shaw	
Siddham	Siddham
Sidd	
SignWriting	SignWriting
Sgnw	
Sinhala	Sinhala
Sinh	
Sogdian	Sogdian
Sogd	
Sora_Sompeng	Sora_Sompeng
Sora	
Soyombo	Soyombo
Soyo	
Sundanese	Sundanese
Sund	
Syloti_Nagri	Syloti_Nagri
Sylo	
Syriac	Syriac
Syrc	
Tagalog	Tagalog
Tglg	

Tagbanwa	Tagbanwa
Tagb	
Tai_Le	Tai_Le
Tale	
Tai_Tham	Tai_Tham
Lana	
Tai_Viet	Tai_Viet
Tavt	
Takri	Takri
Takr	
Tamil	Tamil
Taml	
Tangut	Tangut
Tang	
Telugu	Telugu
Telu	
Thaana	Thaana
Thaa	
Thai	Thai
Tibetan	Tibetan
Tibt	
Tifinagh	Tifinagh
Tfng	
Tirhuta	Tirhuta
Tirh	
Ugaritic	Ugaritic
Ugar	
Vai	Vai
Vaii	
Wancho	Wancho
Wcho	
Warang_Citi	Warang_Citi
Wara	
Yezidi	Yezidi
Yezi	
Yi	Yi

Yiii	
Zanabazar_Square	Zanabazar_Square
Zanb	

22.2.2.9 AtomEscape

With parameter *direction*.

The production *AtomEscape* :: *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an integer *n*.
2. Assert: $n \leq N_{\text{capturingParens}}$.
3. Return ! *BackreferenceMatcher*(*n*, *direction*).

The production *AtomEscape* :: *CharacterEscape* evaluates as follows:

1. Evaluate *CharacterEscape* to obtain a character *ch*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Return ! *CharacterSetMatcher*(*A*, false, *direction*).

The production *AtomEscape* :: *CharacterClassEscape* evaluates as follows:

1. Evaluate *CharacterClassEscape* to obtain a CharSet *A*.
2. Return ! *CharacterSetMatcher*(*A*, false, *direction*).

NOTE

An escape sequence of the form `\` followed by a non-zero decimal number *n* matches the result of the *n*th set of capturing parentheses (22.2.2.1). It is an error if the regular expression has fewer than *n* capturing parentheses. If the regular expression has *n* or more capturing parentheses but the *n*th one is **undefined** because it has not captured anything, then the backreference always succeeds.

The production *AtomEscape* :: **k** *GroupName* evaluates as follows:

1. Search the enclosing *Pattern* for an instance of a *GroupSpecifier* containing a *RegExpIdentifierName* which has a *CapturingGroupName* equal to the *CapturingGroupName* of the *RegExpIdentifierName* contained in *GroupName*.
2. Assert: A unique such *GroupSpecifier* is found.
3. Let *parenIndex* be the number of left-capturing parentheses in the entire regular expression that occur to the left of the located *GroupSpecifier*. This is the total number of *Atom* :: (*GroupSpecifier Disjunction*) Parse Nodes prior to or enclosing the located *GroupSpecifier*, including its immediately enclosing *Atom*.
4. Return ! *BackreferenceMatcher*(*parenIndex*, *direction*).

22.2.2.9.1 BackreferenceMatcher (*n*, *direction*)

The abstract operation *BackreferenceMatcher* takes arguments *n* (a positive integer) and *direction* (1 or -1). It performs the following steps when called:

1. Assert: $n \geq 1$.
2. Return a new *Matcher* with parameters (*x*, *c*) that captures *n* and *direction* and performs the following steps

when called:

- a. **Assert:** x is a State.
- b. **Assert:** c is a Continuation.
- c. Let cap be x 's *captures List*.
- d. Let s be $cap[n]$.
- e. If s is **undefined**, return $c(x)$.
- f. Let e be x 's *endIndex*.
- g. Let len be the number of elements in s .
- h. Let f be $e + direction \times len$.
- i. If $f < 0$ or $f > InputLength$, return failure.
- j. Let g be $\min(e, f)$.
- k. If there exists an integer i between 0 (inclusive) and len (exclusive) such that $Canonicalize(s[i])$ is not the same character value as $Canonicalize(Input[g + i])$, return failure.
- l. Let y be the State (f, cap) .
- m. Return $c(y)$.

22.2.2.10 CharacterEscape

The *CharacterEscape* productions evaluate as follows:

CharacterEscape ::

ControlEscape

c *ControlLetter*

0 [lookahead \notin *DecimalDigit*]

HexEscapeSequence

RegExpUnicodeEscapeSequence

IdentityEscape

1. Let cv be the **CharacterValue** of this *CharacterEscape*.
2. Return the character whose character value is cv .

22.2.2.11 DecimalEscape

The *DecimalEscape* productions evaluate as follows:

DecimalEscape :: *NonZeroDigit* *DecimalDigits*_{opt}

1. Return the **CapturingGroupNumber** of this *DecimalEscape*.

NOTE

If \backslash 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.

22.2.2.12 CharacterClassEscape

The production *CharacterClassEscape* :: **d** evaluates as follows:

1. Return the ten-element CharSet containing the characters **0** through **9** inclusive.

The production *CharacterClassEscape* :: **D** evaluates as follows:

1. Return the CharSet containing all characters not in the CharSet returned by *CharacterClassEscape* :: **d** .

The production *CharacterClassEscape* :: **s** evaluates as follows:

1. Return the CharSet containing all characters corresponding to a code point on the right-hand side of the *WhiteSpace* or *LineTerminator* productions.

The production *CharacterClassEscape* :: **S** evaluates as follows:

1. Return the CharSet containing all characters not in the CharSet returned by *CharacterClassEscape* :: **s** .

The production *CharacterClassEscape* :: **w** evaluates as follows:

1. Return *WordCharacters*.

The production *CharacterClassEscape* :: **W** evaluates as follows:

1. Return the CharSet containing all characters not in the CharSet returned by *CharacterClassEscape* :: **w** .

The production *CharacterClassEscape* :: **P**{ *UnicodePropertyValueExpression* } evaluates as follows:

1. Return the CharSet containing all Unicode code points included in the CharSet returned by *UnicodePropertyValueExpression*.

The production *CharacterClassEscape* :: **P**! { *UnicodePropertyValueExpression* } evaluates as follows:

1. Return the CharSet containing all Unicode code points not included in the CharSet returned by *UnicodePropertyValueExpression*.

The production *UnicodePropertyValueExpression* :: *UnicodePropertyName* = *UnicodePropertyValue* evaluates as follows:

1. Let *ps* be *SourceText* of *UnicodePropertyName*.
2. Let *p* be ! *UnicodeMatchProperty*(*ps*).
3. **Assert:** *p* is a Unicode *property name* or property alias listed in the “*Property name and aliases*” column of *Table 56*.
4. Let *vs* be *SourceText* of *UnicodePropertyValue*.
5. Let *v* be ! *UnicodeMatchPropertyValue*(*p*, *vs*).
6. Return the CharSet containing all Unicode code points whose character database definition includes the property *p* with value *v*.

The production *UnicodePropertyValueExpression* :: *LoneUnicodePropertyNameOrValue* evaluates as follows:

1. Let *s* be *SourceText* of *LoneUnicodePropertyNameOrValue*.
2. If ! *UnicodeMatchPropertyValue*(**General_Category**, *s*) is identical to a *List* of Unicode code points that is the name of a Unicode general category or general category alias listed in the “*Property value and aliases*” column of *Table 58*, then
 - a. Return the CharSet containing all Unicode code points whose character database definition includes the property “*General_Category*” with value *s*.
3. Let *p* be ! *UnicodeMatchProperty*(*s*).
4. **Assert:** *p* is a binary Unicode property or binary property alias listed in the “*Property name and aliases*” column of *Table 57*.

5. Return the CharSet containing all Unicode code points whose character database definition includes the property p with value “True”.

22.2.2.13 CharacterClass

The production $CharacterClass :: [ClassRanges]$ evaluates as follows:

1. Evaluate $ClassRanges$ to obtain a CharSet A .
2. Return the two results A and **false**.

The production $CharacterClass :: [^ ClassRanges]$ evaluates as follows:

1. Evaluate $ClassRanges$ to obtain a CharSet A .
2. Return the two results A and **true**.

22.2.2.14 ClassRanges

The production $ClassRanges :: [empty]$ evaluates as follows:

1. Return the empty CharSet.

The production $ClassRanges :: NonemptyClassRanges$ evaluates as follows:

1. Return the CharSet that is the result of evaluating $NonemptyClassRanges$.

22.2.2.15 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. Let D be $!CharacterRange(A, B)$.
5. Return the union of D and C .

22.2.2.15.1 CharacterRange (A, B)

The abstract operation $CharacterRange$ takes arguments A (a CharSet) and B (a CharSet). It performs the following steps when called:

1. **Assert:** A and B each contain exactly one character.
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. Assert: $i \leq j$.
7. Return the CharSet containing all characters with a character value greater than or equal to i and less than or equal to j .

22.2.2.16 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. Let D be !CharacterRange(A, B).
5. Return the union of D and C .

NOTE 1 $ClassRanges$ can expand into a single $ClassAtom$ and/or ranges of two $ClassAtom$ 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 $\backslash w$) 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.

22.2.2.17 ClassAtom

The production $ClassAtom :: -$ evaluates as follows:

1. Return the CharSet containing the single character – U+002D (HYPHEN-MINUS).

The production *ClassAtom* :: *ClassAtomNoDash* evaluates as follows:

1. Return the CharSet that is the result of evaluating *ClassAtomNoDash*.

22.2.2.18 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*.

22.2.2.19 ClassEscape

The *ClassEscape* productions evaluate as follows:

ClassEscape :: **b**

ClassEscape :: -

ClassEscape :: *CharacterEscape*

1. Let *cv* be the **CharacterValue** of this *ClassEscape*.
2. Let *c* be the character whose character value is *cv*.
3. Return the CharSet containing the single character *c*.

ClassEscape :: *CharacterClassEscape*

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.

22.2.3 The RegExp Constructor

The RegExp **constructor**:

- is `%RegExp%`.
- is the initial value of the "RegExp" property of the **global object**.
- creates and initializes a new RegExp object when called as a function rather than as a **constructor**. Thus the function call **RegExp(...)** is equivalent to the object creation expression **new RegExp(...)** with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified RegExp behaviour must include a **super** call to the RegExp **constructor** to create and initialize subclass instances with the necessary internal slots.

22.2.3.1 RegExp (*pattern*, *flags*)

The following steps are taken:

1. Let *patternIsRegExp* be ? **IsRegExp**(*pattern*).
2. If *NewTarget* is **undefined**, then
 - a. Let *newTarget* be the **active function object**.
 - b. If *patternIsRegExp* is **true** and *flags* is **undefined**, then
 - i. Let *patternConstructor* be ? **Get**(*pattern*, "constructor").
 - ii. If **SameValue**(*newTarget*, *patternConstructor*) is **true**, return *pattern*.
3. Else, let *newTarget* be *NewTarget*.
4. If **Type**(*pattern*) is **Object** and *pattern* has a **[[RegExpMatcher]]** internal slot, then
 - a. Let *P* be *pattern*.**[[OriginalSource]]**.
 - b. If *flags* is **undefined**, let *F* be *pattern*.**[[OriginalFlags]]**.
 - c. Else, let *F* be *flags*.
5. Else if *patternIsRegExp* is **true**, then
 - a. Let *P* be ? **Get**(*pattern*, "source").
 - b. If *flags* is **undefined**, then
 - i. Let *F* be ? **Get**(*pattern*, "flags").
 - c. Else, let *F* be *flags*.
6. Else,
 - a. Let *P* be *pattern*.
 - b. Let *F* be *flags*.
7. Let *O* be ? **RegExpAlloc**(*newTarget*).
8. Return ? **RegExpInitialize**(*O*, *P*, *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 U+005C (REVERSE SOLIDUS) code points must be escaped within the *StringLiteral* to prevent them being removed when the contents of the *StringLiteral* are formed.

22.2.3.2 Abstract Operations for the RegExp Constructor

22.2.3.2.1 RegExpAlloc (*newTarget*)

The abstract operation **RegExpAlloc** takes argument *newTarget*. It performs the following steps when called:

1. Let *obj* be ? **OrdinaryCreateFromConstructor**(*newTarget*, "%**RegExp.prototype**", « **[[RegExpMatcher]]**, **[[OriginalSource]]**, **[[OriginalFlags]]** »).
2. Perform ! **DefinePropertyOrThrow**(*obj*, "lastIndex", PropertyDescriptor { **[[Writable]]**: **true**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **false** }).
3. Return *obj*.

22.2.3.2.2 RegExpInitialize (*obj*, *pattern*, *flags*)

The abstract operation **RegExpInitialize** takes arguments *obj*, *pattern*, and *flags*. It performs the following steps when called:

1. If *pattern* is **undefined**, let *P* be the empty String.
2. Else, let *P* be ? ToString(*pattern*).
3. If *flags* is **undefined**, let *F* be the empty String.
4. Else, let *F* be ? ToString(*flags*).
5. If *F* contains any code unit other than "g", "i", "m", "s", "u", or "y" or if it contains the same code unit more than once, throw a **SyntaxError** exception.
6. If *F* contains "u", let *u* be **true**; else let *u* be **false**.
7. If *u* is **true**, then
 - a. Let *patternText* be ! StringToCodePoints(*P*).
 - b. Let *patternCharacters* be a List whose elements are the code points of *patternText*.
8. Else,
 - a. Let *patternText* be the result of interpreting each of *P*'s 16-bit elements as a Unicode BMP code point. UTF-16 decoding is not applied to the elements.
 - b. Let *patternCharacters* be a List whose elements are the code unit elements of *P*.
9. Let *parseResult* be ParsePattern(*patternText*, *u*).
10. If *parseResult* is a non-empty List of **SyntaxError** objects, throw a **SyntaxError** exception.
11. **Assert**: *parseResult* is a Parse Node for *Pattern*.
12. Set *obj*.[[OriginalSource]] to *P*.
13. Set *obj*.[[OriginalFlags]] to *F*.
14. Set *obj*.[[RegExpMatcher]] to the **Abstract Closure** that evaluates *parseResult* by applying the semantics provided in 22.2.2 using *patternCharacters* as the pattern's List of *SourceCharacter* values and *F* as the flag parameters.
15. Perform ? Set(*obj*, "lastIndex", +0_F, **true**).
16. Return *obj*.

22.2.3.2.3 Static Semantics: ParsePattern (*patternText*, *u*)

The abstract operation ParsePattern takes arguments *patternText* (a sequence of Unicode code points) and *u* (a Boolean). It performs the following steps when called:

1. If *u* is **true**, then
 - a. Let *parseResult* be ParseText(*patternText*, Pattern_[+U, +N]).
2. Else,
 - a. Let *parseResult* be ParseText(*patternText*, Pattern_[~U, ~N]).
 - b. If *parseResult* is a Parse Node and *parseResult* contains a *GroupName*, then
 - i. Set *parseResult* to ParseText(*patternText*, Pattern_[~U, +N]).
3. Return *parseResult*.

22.2.3.2.4 RegExpCreate (*P*, *F*)

The abstract operation RegExpCreate takes arguments *P* and *F*. It performs the following steps when called:

1. Let *obj* be ? RegExpAlloc(%RegExp%).
2. Return ? RegExpInitialize(*obj*, *P*, *F*).

22.2.3.2.5 EscapeRegExpPattern (*P*, *F*)

The abstract operation EscapeRegExpPattern takes arguments *P* and *F*. It performs the following steps when called:

1. Let *S* be a String in the form of a *Pattern*_[~U] (*Pattern*_[+U] if *F* contains "u") equivalent to *P* interpreted as UTF-16 encoded Unicode code points (6.1.4), in which certain code points are escaped as described below. *S* may or may not be identical to *P*; however, the **Abstract Closure** that would result from evaluating *S* as a *Pattern*_[~U] (*Pattern*_[+U] if *F* contains "u") must behave identically to the **Abstract Closure** 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-concatenation** of `"/", S, "/"`, and *F* can be parsed (in an appropriate lexical context) as a *RegularExpressionLiteral* that behaves identically to the constructed regular expression. For example, if *P* is `/`, then *S* could be `\"/` or `\"u002F`, among other possibilities, but not `/`, because `///` followed by *F* would be parsed as a *SingleLineComment* rather than a *RegularExpressionLiteral*. If *P* is the empty String, this specification can be met by letting *S* be `"(?:)"`.
3. Return *S*.

22.2.4 Properties of the RegExp Constructor

The RegExp **constructor**:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

22.2.4.1 RegExp.prototype

The initial value of **RegExp.prototype** is the **RegExp prototype object**.

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

22.2.4.2 get RegExp [@@species]

RegExp[@@species] is an **accessor property** whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the **"name"** property of this function is `"get [Symbol.species]"`.

NOTE RegExp prototype methods normally use their **this** value's **constructor** to create a derived object. However, a subclass **constructor** may over-ride that default behaviour by redefining its `@@species` property.

22.2.5 Properties of the RegExp Prototype Object

The *RegExp prototype object*:

- is `%RegExp.prototype%`.
- is an **ordinary object**.
- is not a RegExp instance and does not have a `[[RegExpMatcher]]` internal slot or any of the other internal slots of RegExp instance objects.

- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.

NOTE The `RegExp` prototype object does not have a `"valueOf"` property of its own; however, it inherits the `"valueOf"` property from the `Object` prototype object.

22.2.5.1 `RegExp.prototype.constructor`

The initial value of `RegExp.prototype.constructor` is `%RegExp%`.

22.2.5.2 `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. Perform `? RequireInternalSlot(R, [[RegExpMatcher]])`.
3. Let *S* be `? ToString(string)`.
4. Return `? RegExpBuiltinExec(R, S)`.

22.2.5.2.1 `RegExpExec (R, S)`

The abstract operation `RegExpExec` takes arguments *R* and *S*. It performs the following steps when called:

1. **Assert:** `Type(R)` is Object.
2. **Assert:** `Type(S)` is String.
3. Let *exec* be `? Get(R, "exec")`.
4. If `IsCallable(exec)` is **true**, then
 - a. Let *result* be `? Call(exec, R, « S »)`.
 - b. If `Type(result)` is neither Object nor Null, throw a **TypeError** exception.
 - c. Return *result*.
5. Perform `? RequireInternalSlot(R, [[RegExpMatcher]])`.
6. Return `? RegExpBuiltinExec(R, S)`.

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"`.

22.2.5.2.2 `RegExpBuiltinExec (R, S)`

The abstract operation `RegExpBuiltinExec` takes arguments *R* and *S*. It performs the following steps when called:

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 $\mathbb{R}(? ToLength(? Get(R, "lastIndex")))$.

5. Let *flags* be *R*.[[OriginalFlags]].
6. If *flags* contains "g", let *global* be **true**; else let *global* be **false**.
7. If *flags* contains "y", let *sticky* be **true**; else let *sticky* be **false**.
8. If *global* is **false** and *sticky* is **false**, set *lastIndex* to 0.
9. Let *matcher* be *R*.[[RegExpMatcher]].
10. If *flags* contains "u", let *fullUnicode* be **true**; else let *fullUnicode* be **false**.
11. Let *matchSucceeded* be **false**.
12. Repeat, while *matchSucceeded* is **false**,
 - a. If *lastIndex* > *length*, then
 - i. If *global* is **true** or *sticky* is **true**, then
 1. Perform ? Set(*R*, "lastIndex", +0_F, **true**).
 - ii. Return **null**.
 - b. Let *r* be *matcher*(*S*, *lastIndex*).
 - c. If *r* is failure, then
 - i. If *sticky* is **true**, then
 1. Perform ? Set(*R*, "lastIndex", +0_F, **true**).
 2. Return **null**.
 - ii. Set *lastIndex* to AdvanceStringIndex(*S*, *lastIndex*, *fullUnicode*).
 - d. Else,
 - i. Assert: *r* is a State.
 - ii. Set *matchSucceeded* to **true**.
13. Let *e* be *r*'s *endIndex* value.
14. If *fullUnicode* is **true**, then
 - a. *e* is an index into the *Input* character list, derived from *S*, matched by *matcher*. Let *eUTF* be the smallest index into *S* that corresponds to the character at element *e* of *Input*. If *e* is greater than or equal to the number of elements in *Input*, then *eUTF* is the number of code units in *S*.
 - b. Set *e* to *eUTF*.
15. If *global* is **true** or *sticky* is **true**, then
 - a. Perform ? Set(*R*, "lastIndex", *F*(*e*), **true**).
16. Let *n* be the number of elements in *r*'s *captures List*. (This is the same value as 22.2.2.1's *NcapturingParens*.)
17. Assert: $n < 2^{32} - 1$.
18. Let *A* be ! ArrayCreate(*n* + 1).
19. Assert: The mathematical value of *A*'s "length" property is *n* + 1.
20. Perform ! CreateDataPropertyOrThrow(*A*, "index", *F*(*lastIndex*)).
21. Perform ! CreateDataPropertyOrThrow(*A*, "input", *S*).
22. Let *matchedSubstr* be the substring of *S* from *lastIndex* to *e*.
23. Perform ! CreateDataPropertyOrThrow(*A*, "0", *matchedSubstr*).
24. If *R* contains any *GroupName*, then
 - a. Let *groups* be ! OrdinaryObjectCreate(**null**).
25. Else,
 - a. Let *groups* be **undefined**.
26. Perform ! CreateDataPropertyOrThrow(*A*, "groups", *groups*).
27. For each integer *i* such that $i \geq 1$ and $i \leq n$, do
 - a. Let *captureI* be *i*th element of *r*'s *captures List*.
 - b. If *captureI* is **undefined**, let *capturedValue* be **undefined**.
 - c. Else if *fullUnicode* is **true**, then
 - i. Assert: *captureI* is a List of code points.

- ii. Let *capturedValue* be ! `CodePointsToString(captureI)`.
 - d. Else,
 - i. Assert: *fullUnicode* is **false**.
 - ii. Assert: *captureI* is a `List` of code units.
 - iii. Let *capturedValue* be the `String` value consisting of the code units of *captureI*.
 - e. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(i)), capturedValue)`.
 - f. If the *i*th capture of *R* was defined with a *GroupName*, then
 - i. Let *s* be the `CapturingGroupName` of the corresponding `RegExpIdentifierName`.
 - ii. Perform ! `CreateDataPropertyOrThrow(groups, s, capturedValue)`.
28. Return *A*.

22.2.5.2.3 AdvanceStringIndex (*S*, *index*, *unicode*)

The abstract operation `AdvanceStringIndex` takes arguments *S* (a `String`), *index* (a non-negative `integer`), and *unicode* (a `Boolean`). It performs the following steps when called:

1. Assert: *index* ≤ 2⁵³ - 1.
2. If *unicode* is **false**, return *index* + 1.
3. Let *length* be the number of code units in *S*.
4. If *index* + 1 ≥ *length*, return *index* + 1.
5. Let *cp* be ! `CodePointAt(S, index)`.
6. Return *index* + *cp*.[[`CodeUnitCount`]].

22.2.5.3 get `RegExp.prototype.dotAll`

`RegExp.prototype.dotAll` 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`, throw a **TypeError** exception.
3. If *R* does not have an `[[OriginalFlags]]` internal slot, then
 - a. If `SameValue(R, %RegExp.prototype%)` is **true**, return **undefined**.
 - b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be *R*.[[`OriginalFlags`]].
5. If *flags* contains the code unit 0x0073 (LATIN SMALL LETTER S), return **true**.
6. Return **false**.

22.2.5.4 get `RegExp.prototype.flags`

`RegExp.prototype.flags` 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`, throw a **TypeError** exception.
3. Let *result* be the empty `String`.
4. Let *global* be ! `ToBoolean(? Get(R, "global"))`.
5. If *global* is **true**, append the code unit 0x0067 (LATIN SMALL LETTER G) as the last code unit of *result*.
6. Let *ignoreCase* be ! `ToBoolean(? Get(R, "ignoreCase"))`.
7. If *ignoreCase* is **true**, append the code unit 0x0069 (LATIN SMALL LETTER I) as the last code unit of *result*.

8. Let *multiline* be ! ToBoolean(? Get(*R*, "multiline")).
9. If *multiline* is **true**, append the code unit 0x006D (LATIN SMALL LETTER M) as the last code unit of *result*.
10. Let *dotAll* be ! ToBoolean(? Get(*R*, "dotAll")).
11. If *dotAll* is **true**, append the code unit 0x0073 (LATIN SMALL LETTER S) as the last code unit of *result*.
12. Let *unicode* be ! ToBoolean(? Get(*R*, "unicode")).
13. If *unicode* is **true**, append the code unit 0x0075 (LATIN SMALL LETTER U) as the last code unit of *result*.
14. Let *sticky* be ! ToBoolean(? Get(*R*, "sticky")).
15. If *sticky* is **true**, append the code unit 0x0079 (LATIN SMALL LETTER Y) as the last code unit of *result*.
16. Return *result*.

22.2.5.5 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, throw a **TypeError** exception.
3. If *R* does not have an [[OriginalFlags]] internal slot, then
 - a. If SameValue(*R*, %RegExp.prototype%) is **true**, return **undefined**.
 - b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be *R*.[[OriginalFlags]].
5. If *flags* contains the code unit 0x0067 (LATIN SMALL LETTER G), return **true**.
6. Return **false**.

22.2.5.6 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, throw a **TypeError** exception.
3. If *R* does not have an [[OriginalFlags]] internal slot, then
 - a. If SameValue(*R*, %RegExp.prototype%) is **true**, return **undefined**.
 - b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be *R*.[[OriginalFlags]].
5. If *flags* contains the code unit 0x0069 (LATIN SMALL LETTER I), return **true**.
6. Return **false**.

22.2.5.7 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, throw a **TypeError** exception.
3. Let *S* be ? ToString(*string*).
4. Let *global* be ! ToBoolean(? Get(*rx*, "global")).
5. If *global* is **false**, then
 - a. Return ? RegExpExec(*rx*, *S*).
6. Else,

- a. **Assert**: *global* is **true**.
- b. Let *fullUnicode* be ! **ToBoolean**(? **Get**(*rx*, "unicode")).
- c. Perform ? **Set**(*rx*, "lastIndex", +0_F, **true**).
- d. Let *A* be ! **ArrayCreate**(0).
- e. Let *n* be 0.
- f. Repeat,
 - i. Let *result* be ? **RegExpExec**(*rx*, *S*).
 - ii. If *result* is **null**, then
 1. If *n* = 0, return **null**.
 2. Return *A*.
 - iii. Else,
 1. Let *matchStr* be ? **ToString**(? **Get**(*result*, "0")).
 2. Perform ! **CreateDataPropertyOrThrow**(*A*, ! **ToString**(**F**(*n*)), *matchStr*).
 3. If *matchStr* is the empty String, then
 - a. Let *thisIndex* be **R**(? **ToLength**(? **Get**(*rx*, "lastIndex"))).
 - b. Let *nextIndex* be **AdvanceStringIndex**(*S*, *thisIndex*, *fullUnicode*).
 - c. Perform ? **Set**(*rx*, "lastIndex", **F**(*nextIndex*), **true**).
 4. Set *n* to *n* + 1.

The value of the "name" property of this function is "[Symbol.match]".

NOTE

The `@@match` property is used by the `IsRegExp` abstract operation to identify objects that have the basic behaviour of regular expressions. The absence of a `@@match` property or the existence of such a property whose value does not Boolean coerce to **true** indicates that the object is not intended to be used as a regular expression object.

22.2.5.8 RegExp.prototype [@@matchAll] (*string*)

When the `@@matchAll` method is called with argument *string*, the following steps are taken:

1. Let *R* be the **this** value.
2. If **Type**(*R*) is not Object, throw a **TypeError** exception.
3. Let *S* be ? **ToString**(*string*).
4. Let *C* be ? **SpeciesConstructor**(*R*, %RegExp%).
5. Let *flags* be ? **ToString**(? **Get**(*R*, "flags")).
6. Let *matcher* be ? **Construct**(*C*, « *R*, *flags* »).
7. Let *lastIndex* be ? **ToLength**(? **Get**(*R*, "lastIndex")).
8. Perform ? **Set**(*matcher*, "lastIndex", *lastIndex*, **true**).
9. If *flags* contains "g", let *global* be **true**.
10. Else, let *global* be **false**.
11. If *flags* contains "u", let *fullUnicode* be **true**.
12. Else, let *fullUnicode* be **false**.
13. Return ! **CreateRegExpStringIterator**(*matcher*, *S*, *global*, *fullUnicode*).

The value of the "name" property of this function is "[Symbol.matchAll]".

22.2.5.9 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, throw a **TypeError** exception.
3. If *R* does not have an `[[OriginalFlags]]` internal slot, then
 - a. If **SameValue**(*R*, `%RegExp.prototype%`) is **true**, return **undefined**.
 - b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be *R*.`[[OriginalFlags]]`.
5. If *flags* contains the code unit 0x006D (LATIN SMALL LETTER M), return **true**.
6. Return **false**.

22.2.5.10 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, throw a **TypeError** exception.
3. Let *S* be ? **ToString**(*string*).
4. Let *lengthS* be the number of code unit elements in *S*.
5. Let *functionalReplace* be **IsCallable**(*replaceValue*).
6. If *functionalReplace* is **false**, then
 - a. Set *replaceValue* to ? **ToString**(*replaceValue*).
7. Let *global* be ! **ToBoolean**(? **Get**(*rx*, "global")).
8. If *global* is **true**, then
 - a. Let *fullUnicode* be ! **ToBoolean**(? **Get**(*rx*, "unicode")).
 - b. Perform ? **Set**(*rx*, "lastIndex", +0_F, **true**).
9. Let *results* be a new empty **List**.
10. Let *done* be **false**.
11. Repeat, while *done* is **false**,
 - a. Let *result* be ? **RegExpExec**(*rx*, *S*).
 - b. If *result* is **null**, set *done* to **true**.
 - c. Else,
 - i. Append *result* to the end of *results*.
 - ii. If *global* is **false**, set *done* to **true**.
 - iii. Else,
 1. Let *matchStr* be ? **ToString**(? **Get**(*result*, "0")).
 2. If *matchStr* is the empty String, then
 - a. Let *thisIndex* be **ℝ**(? **ToLength**(? **Get**(*rx*, "lastIndex"))).
 - b. Let *nextIndex* be **AdvanceStringIndex**(*S*, *thisIndex*, *fullUnicode*).
 - c. Perform ? **Set**(*rx*, "lastIndex", **ℱ**(*nextIndex*), **true**).
12. Let *accumulatedResult* be the empty String.
13. Let *nextSourcePosition* be 0.
14. For each element *result* of *results*, do
 - a. Let *resultLength* be ? **LengthOfArrayLike**(*result*).
 - b. Let *nCaptures* be **max**(*resultLength* - 1, 0).
 - c. Let *matched* be ? **ToString**(? **Get**(*result*, "0")).
 - d. Let *matchLength* be the number of code units in *matched*.

- e. Let *position* be ? `ToIntegerOrInfinity(? Get(result, "index"))`.
 - f. Set *position* to the result of `clamping position` between 0 and *lengthS*.
 - g. Let *n* be 1.
 - h. Let *captures* be a new empty `List`.
 - i. Repeat, while $n \leq nCaptures$,
 - i. Let *capN* be ? `Get(result, ! ToString($\mathbb{F}(n)$))`.
 - ii. If *capN* is not **undefined**, then
 1. Set *capN* to ? `ToString(capN)`.
 - iii. Append *capN* as the last element of *captures*.
 - iv. Set *n* to $n + 1$.
 - j. Let *namedCaptures* be ? `Get(result, "groups")`.
 - k. If *functionalReplace* is **true**, then
 - i. Let *replacerArgs* be « *matched* ».
 - ii. Append in `List` order the elements of *captures* to the end of the `List replacerArgs`.
 - iii. Append $\mathbb{F}(position)$ and *S* to *replacerArgs*.
 - iv. If *namedCaptures* is not **undefined**, then
 1. Append *namedCaptures* as the last element of *replacerArgs*.
 - v. Let *replValue* be ? `Call(replaceValue, undefined, replacerArgs)`.
 - vi. Let *replacement* be ? `ToString(replValue)`.
 - l. Else,
 - i. If *namedCaptures* is not **undefined**, then
 1. Set *namedCaptures* to ? `ToObject(namedCaptures)`.
 - ii. Let *replacement* be ? `GetSubstitution(matched, S, position, captures, namedCaptures, replaceValue)`.
 - m. If $position \geq nextSourcePosition$, then
 - i. NOTE: *position* should not normally move backwards. If it does, it is an indication of an 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. Set *accumulatedResult* to the `string-concatenation` of *accumulatedResult*, the `substring` of *S* from *nextSourcePosition* to *position*, and *replacement*.
 - iii. Set *nextSourcePosition* to $position + matchLength$.
15. If $nextSourcePosition \geq lengthS$, return *accumulatedResult*.
16. Return the `string-concatenation` of *accumulatedResult* and the `substring` of *S* from *nextSourcePosition*.

The value of the "name" property of this function is "[Symbol.replace]".

22.2.5.11 `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`, throw a **TypeError** exception.
3. Let *S* be ? `ToString(string)`.
4. Let *previousLastIndex* be ? `Get(rx, "lastIndex")`.
5. If `SameValue(previousLastIndex, +0F)` is **false**, then
 - a. Perform ? `Set(rx, "lastIndex", +0F, true)`.
6. Let *result* be ? `RegExpExec(rx, S)`.
7. Let *currentLastIndex* be ? `Get(rx, "lastIndex")`.
8. If `SameValue(currentLastIndex, previousLastIndex)` is **false**, then

- a. Perform ? `Set(rx, "lastIndex", previousLastIndex, true)`.
9. If `result` is `null`, return `-1`.
10. Return ? `Get(result, "index")`.

The value of the `"name"` property of this function is `"[Symbol.search]"`.

NOTE The `"lastIndex"` and `"global"` properties of this `RegExp` object are ignored when performing the search. The `"lastIndex"` property is left unchanged.

22.2.5.12 `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`, throw a **`TypeError`** exception.
3. If `R` does not have an `[[OriginalSource]]` internal slot, then
 - a. If `SameValue(R, %RegExp.prototype%)` is `true`, return `"(?:)"`.
 - b. Otherwise, throw a **`TypeError`** exception.
4. **Assert:** `R` has an `[[OriginalFlags]]` internal slot.
5. Let `src` be `R.[[OriginalSource]]`.
6. Let `flags` be `R.[[OriginalFlags]]`.
7. Return `EscapeRegExpPattern(src, flags)`.

22.2.5.13 `RegExp.prototype [@@split] (string, limit)`

NOTE 1

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 String 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, the regular expression 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 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 index of the String is considered, even if backtracking could yield a non-empty substring match at that index. (For example, `/a*/[Symbol.split]("ab")` evaluates to the array `["a", "b"]`, while `/a*/[Symbol.split]("ab")` evaluates to the array `["", "b"]`.)

If *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 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,

```
/<(\\)?([^\<]+)>/[Symbol.split]("A<B>bold</B>and<CODE>coded</CODE>")
```

evaluates to the array

```
["A", undefined, "B", "bold", "/", "B", "and", undefined, "CODE", "coded"]
```

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, throw a **TypeError** exception.
3. Let *S* be `? ToString(string)`.
4. Let *C* be `? SpeciesConstructor(rx, %RegExp%)`.
5. Let *flags* be `? ToString(? Get(rx, "flags"))`.
6. If *flags* contains "u", let *unicodeMatching* be **true**.
7. Else, let *unicodeMatching* be **false**.
8. If *flags* contains "y", let *newFlags* be *flags*.
9. Else, let *newFlags* be the string-concatenation of *flags* and "y".
10. Let *splitter* be `? Construct(C, « rx, newFlags »)`.
11. Let *A* be `! ArrayCreate(0)`.
12. Let *lengthA* be 0.
13. If *limit* is **undefined**, let *lim* be $2^{32} - 1$; else let *lim* be $\mathbb{R}(? ToUint32(limit))$.
14. If *lim* is 0, return *A*.
15. Let *size* be the length of *S*.

16. If *size* is 0, then
 - a. Let *z* be ? `RegExpExec(splitter, S)`.
 - b. If *z* is not **null**, return *A*.
 - c. Perform ! `CreateDataPropertyOrThrow(A, "0", S)`.
 - d. Return *A*.
17. Let *p* be 0.
18. Let *q* be *p*.
19. Repeat, while *q* < *size*,
 - a. Perform ? `Set(splitter, "lastIndex", F(q), true)`.
 - b. Let *z* be ? `RegExpExec(splitter, S)`.
 - c. If *z* is **null**, set *q* to `AdvanceStringIndex(S, q, unicodeMatching)`.
 - d. Else,
 - i. Let *e* be `R(? ToLength(? Get(splitter, "lastIndex")))`.
 - ii. Set *e* to `min(e, size)`.
 - iii. If *e* = *p*, set *q* to `AdvanceStringIndex(S, q, unicodeMatching)`.
 - iv. Else,
 1. Let *T* be the **substring** of *S* from *p* to *q*.
 2. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(lengthA)), T)`.
 3. Set *lengthA* to *lengthA* + 1.
 4. If *lengthA* = *lim*, return *A*.
 5. Set *p* to *e*.
 6. Let *numberOfCaptures* be ? `LengthOfArrayLike(z)`.
 7. Set *numberOfCaptures* to `max(numberOfCaptures - 1, 0)`.
 8. Let *i* be 1.
 9. Repeat, while *i* ≤ *numberOfCaptures*,
 - a. Let *nextCapture* be ? `Get(z, ! ToString(F(i)))`.
 - b. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(lengthA)), nextCapture)`.
 - c. Set *i* to *i* + 1.
 - d. Set *lengthA* to *lengthA* + 1.
 - e. If *lengthA* = *lim*, return *A*.
 10. Set *q* to *p*.
 20. Let *T* be the **substring** of *S* from *p* to *size*.
 21. Perform ! `CreateDataPropertyOrThrow(A, ! ToString(F(lengthA)), T)`.
 22. Return *A*.

The value of the "name" property of this function is "[Symbol.split]".

NOTE 2 The **@@split** method ignores the value of the "global" and "sticky" properties of this RegExp object.

22.2.5.14 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, throw a **TypeError** exception.
3. If *R* does not have an `[[OriginalFlags]]` internal slot, then

- a. If `SameValue(R, %RegExp.prototype%)` is **true**, return **undefined**.
- b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be `R.[[OriginalFlags]]`.
5. If *flags* contains the code unit 0x0079 (LATIN SMALL LETTER Y), return **true**.
6. Return **false**.

22.2.5.15 `RegExp.prototype.test (S)`

The following steps are taken:

1. Let *R* be the **this** value.
2. If `Type(R)` is not `Object`, throw a **TypeError** exception.
3. Let *string* be `? ToString(S)`.
4. Let *match* be `? RegExpExec(R, string)`.
5. If *match* is not **null**, return **true**; else return **false**.

22.2.5.16 `RegExp.prototype.toString ()`

1. Let *R* be the **this** value.
2. If `Type(R)` is not `Object`, throw a **TypeError** exception.
3. Let *pattern* be `? ToString(? Get(R, "source"))`.
4. Let *flags* be `? ToString(? Get(R, "flags"))`.
5. Let *result* be the *string-concatenation* of `"/"`, *pattern*, `"/"`, and *flags*.
6. 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.

22.2.5.17 `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`, throw a **TypeError** exception.
3. If *R* does not have an `[[OriginalFlags]]` internal slot, then
 - a. If `SameValue(R, %RegExp.prototype%)` is **true**, return **undefined**.
 - b. Otherwise, throw a **TypeError** exception.
4. Let *flags* be `R.[[OriginalFlags]]`.
5. If *flags* contains the code unit 0x0075 (LATIN SMALL LETTER U), return **true**.
6. Return **false**.

22.2.6 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 *Abstract Closure* representation of the *Pattern* of the `RegExp` object.

NOTE

Prior to ECMAScript 2015, 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:

22.2.6.1 lastIndex

The value of the "lastIndex" property specifies the String index at which to start the next match. It is coerced to an [integral Number](#) when used (see 22.2.5.2.2). This property shall have the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

22.2.7 RegExp String Iterator Objects

A RegExp String Iterator is an object, that represents a specific iteration over some specific String instance object, matching against some specific RegExp instance object. There is not a named [constructor](#) for RegExp String Iterator objects. Instead, RegExp String Iterator objects are created by calling certain methods of RegExp instance objects.

22.2.7.1 CreateRegExpStringIterator (*R*, *S*, *global*, *fullUnicode*)

The abstract operation CreateRegExpStringIterator takes arguments *R*, *S*, *global*, and *fullUnicode*. It performs the following steps when called:

1. **Assert:** Type(*S*) is String.
2. **Assert:** Type(*global*) is Boolean.
3. **Assert:** Type(*fullUnicode*) is Boolean.
4. Let *closure* be a new [Abstract Closure](#) with no parameters that captures *R*, *S*, *global*, and *fullUnicode* and performs the following steps when called:
 - a. Repeat,
 - i. Let *match* be ? [RegExpExec](#)(*R*, *S*).
 - ii. If *match* is **null**, return **undefined**.
 - iii. If *global* is **false**, then
 1. Perform ? [Yield](#)(*match*).
 2. Return **undefined**.
 - iv. Let *matchStr* be ? [ToString](#)(? [Get](#)(*match*, "0")).
 - v. If *matchStr* is the empty String, then
 1. Let *thisIndex* be \mathbb{R} (? [ToLength](#)(? [Get](#)(*R*, "lastIndex"))).
 2. Let *nextIndex* be ! [AdvanceStringIndex](#)(*S*, *thisIndex*, *fullUnicode*).
 3. Perform ? [Set](#)(*R*, "lastIndex", \mathbb{F} (*nextIndex*), **true**).
 - vi. Perform ? [Yield](#)(*match*).
5. Return ! [CreateIteratorFromClosure](#)(*closure*, "%RegExpStringIteratorPrototype%", %RegExpStringIteratorPrototype%).

22.2.7.2 The %RegExpStringIteratorPrototype% Object

The %*RegExpStringIteratorPrototype*% object:

- has properties that are inherited by all RegExp String Iterator Objects.

- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%IteratorPrototype%`.
- has the following properties:

22.2.7.2.1 `%RegExpStringIteratorPrototype%.next ()`

1. Return ? [GeneratorResume](#)(`this` value, empty, `"%RegExpStringIteratorPrototype%"`).

22.2.7.2.2 `%RegExpStringIteratorPrototype%` [`@@toStringTag`]

The initial value of the `@@toStringTag` property is the String value `"RegExp String Iterator"`.

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

23 Indexed Collections

23.1 Array Objects

Array objects are exotic objects that give special treatment to a certain class of property names. See [10.4.2](#) for a definition of this special treatment.

23.1.1 The Array Constructor

The Array [constructor](#):

- is `%Array%`.
- is the initial value of the `"Array"` property of the [global object](#).
- creates and initializes a new [Array exotic object](#) when called as a [constructor](#).
- also creates and initializes a new Array object when called as a function rather than as a [constructor](#). Thus the function call `Array(...)` is equivalent to the object creation expression `new Array(...)` with the same arguments.
- is a function whose behaviour differs based upon the number and types of its arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the exotic Array behaviour must include a **super** call to the Array [constructor](#) to initialize subclass instances that are Array exotic objects. However, most of the **Array.prototype** methods are generic methods that are not dependent upon their `this` value being an [Array exotic object](#).
- has a `"length"` property whose value is `1F`.

23.1.1.1 `Array (...values)`

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

1. If `NewTarget` is **undefined**, let `newTarget` be the [active function object](#); else let `newTarget` be `NewTarget`.
2. Let `proto` be ? [GetPrototypeFromConstructor](#)(`newTarget`, `"%Array.prototype%"`).
3. Let `numberOfArgs` be the number of elements in `values`.

4. If *numberOfArgs* = 0, then
 - a. Return ! **ArrayCreate**(0, *proto*).
5. Else if *numberOfArgs* = 1, then
 - a. Let *len* be *values*[0].
 - b. Let *array* be ! **ArrayCreate**(0, *proto*).
 - c. If **Type**(*len*) is not **Number**, then
 - i. Perform ! **CreateDataPropertyOrThrow**(*array*, "0", *len*).
 - ii. Let *intLen* be 1_F.
 - d. Else,
 - i. Let *intLen* be ! **ToUint32**(*len*).
 - ii. If *intLen* is not the same value as *len*, throw a **RangeError** exception.
 - e. Perform ! **Set**(*array*, "length", *intLen*, **true**).
 - f. Return *array*.
6. Else,
 - a. **Assert**: *numberOfArgs* ≥ 2.
 - b. Let *array* be ? **ArrayCreate**(*numberOfArgs*, *proto*).
 - c. Let *k* be 0.
 - d. Repeat, while *k* < *numberOfArgs*,
 - i. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - ii. Let *itemK* be *values*[*k*].
 - iii. Perform ! **CreateDataPropertyOrThrow**(*array*, *Pk*, *itemK*).
 - iv. Set *k* to *k* + 1.
 - e. **Assert**: The mathematical value of *array*'s "length" property is *numberOfArgs*.
 - f. Return *array*.

23.1.2 Properties of the Array Constructor

The Array **constructor**:

- has a **[[Prototype]]** internal slot whose value is `%Function.prototype%`.
- has the following properties:

23.1.2.1 **Array.from** (*items* [, *mapfn* [, *thisArg*]])

When the **from** method is called with argument *items* and optional arguments *mapfn* and *thisArg*, the following steps are taken:

1. Let *C* be the **this** value.
2. If *mapfn* is **undefined**, let *mapping* be **false**.
3. Else,
 - a. If **IsCallable**(*mapfn*) is **false**, throw a **TypeError** exception.
 - b. Let *mapping* be **true**.
4. Let *usingIterator* be ? **GetMethod**(*items*, @@iterator).
5. If *usingIterator* is not **undefined**, then
 - a. If **IsConstructor**(*C*) is **true**, then
 - i. Let *A* be ? **Construct**(*C*).
 - b. Else,
 - i. Let *A* be ! **ArrayCreate**(0).

- c. Let *iteratorRecord* be ? `GetIterator(items, sync, usingIterator)`.
- d. Let *k* be 0.
- e. Repeat,
 - i. If $k \geq 2^{53} - 1$, then
 1. Let *error* be `ThrowCompletion`(a newly created `TypeError` object).
 2. Return ? `IteratorClose(iteratorRecord, error)`.
 - ii. Let *Pk* be ! `ToString`($\mathbb{F}(k)$).
 - iii. Let *next* be ? `IteratorStep(iteratorRecord)`.
 - iv. If *next* is **false**, then
 1. Perform ? `Set(A, "length", $\mathbb{F}(k)$, true)`.
 2. Return *A*.
 - v. Let *nextValue* be ? `IteratorValue(next)`.
 - vi. If *mapping* is **true**, then
 1. Let *mappedValue* be `Call(mapfn, thisArg, « nextValue, $\mathbb{F}(k)$ »)`.
 2. If *mappedValue* is an abrupt completion, return ? `IteratorClose(iteratorRecord, mappedValue)`.
 3. Set *mappedValue* to *mappedValue*.[[Value]].
 - vii. Else, let *mappedValue* be *nextValue*.
 - viii. Let *defineStatus* be `CreateDataPropertyOrThrow(A, Pk, mappedValue)`.
 - ix. If *defineStatus* is an abrupt completion, return ? `IteratorClose(iteratorRecord, defineStatus)`.
 - x. Set *k* to *k* + 1.
6. NOTE: *items* is not an Iterable so assume it is an array-like object.
7. Let *arrayLike* be ! `ToObject(items)`.
8. Let *len* be ? `LengthOfArrayLike(arrayLike)`.
9. If `IsConstructor(C)` is **true**, then
 - a. Let *A* be ? `Construct(C, « $\mathbb{F}(len)$ »)`.
10. Else,
 - a. Let *A* be ? `ArrayCreate(len)`.
11. Let *k* be 0.
12. Repeat, while $k < len$,
 - a. Let *Pk* be ! `ToString`($\mathbb{F}(k)$).
 - b. Let *kValue* be ? `Get(arrayLike, Pk)`.
 - c. If *mapping* is **true**, then
 - i. Let *mappedValue* be ? `Call(mapfn, thisArg, « kValue, $\mathbb{F}(k)$ »)`.
 - d. Else, let *mappedValue* be *kValue*.
 - e. Perform ? `CreateDataPropertyOrThrow(A, Pk, mappedValue)`.
 - f. Set *k* to *k* + 1.
13. Perform ? `Set(A, "length", $\mathbb{F}(len)$, true)`.
14. Return *A*.

NOTE

The **from** function is an intentionally generic factory method; it does not require that its **this** value be the Array `constructor`. Therefore it can be transferred to or inherited by any other constructors that may be called with a single numeric argument.

23.1.2.2 Array.isArray (*arg*)

The **isArray** function takes one argument *arg*, and performs the following steps:

1. Return ? `IsArray(arg)`.

23.1.2.3 `Array.of (...items)`

When the **of** method is called with any number of arguments, the following steps are taken:

1. Let *len* be the number of elements in *items*.
2. Let *lenNumber* be `F(len)`.
3. Let *C* be the **this** value.
4. If `IsConstructor(C)` is **true**, then
 - a. Let *A* be ? `Construct(C, « lenNumber »)`.
5. Else,
 - a. Let *A* be ? `ArrayCreate(len)`.
6. Let *k* be 0.
7. Repeat, while *k* < *len*,
 - a. Let *kValue* be *items*[*k*].
 - b. Let *Pk* be ! `ToString(F(k))`.
 - c. Perform ? `CreateDataPropertyOrThrow(A, Pk, kValue)`.
 - d. Set *k* to *k* + 1.
8. Perform ? `Set(A, "length", lenNumber, true)`.
9. Return *A*.

NOTE The **of** function is an intentionally generic factory method; it does not require that its **this** value be the `Array` constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument.

23.1.2.4 `Array.prototype`

The value of **Array.prototype** is the `Array` prototype object.

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

23.1.2.5 `get Array [@@species]`

Array[@@species] is an `accessor property` whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the **"name"** property of this function is **"get [Symbol.species]"**.

NOTE `Array` prototype methods normally use their **this** value's `constructor` to create a derived object. However, a subclass `constructor` may over-ride that default behaviour by redefining its `@@species` property.

23.1.3 Properties of the Array Prototype Object

The *Array* prototype object:

- is *%Array.prototype%*.
- is an **Array exotic object** and has the internal methods specified for such objects.
- has a **"length"** property whose initial value is $+0_{\text{F}}$ and whose attributes are { **[[Writable]]**: **true**, **[[Enumerable]]**: **false**, **[[Configurable]]**: **false** }.
- has a **[[Prototype]]** internal slot whose value is *%Object.prototype%*.

NOTE The *Array* prototype object is specified to be an **Array exotic object** to ensure compatibility with ECMAScript code that was created prior to the ECMAScript 2015 specification.

23.1.3.1 *Array.prototype.concat* (...*items*)

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.

The following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *A* be ? **ArraySpeciesCreate**(*O*, 0).
3. Let *n* be 0.
4. Prepend *O* to *items*.
5. For each element *E* of *items*, do
 - a. Let *spreadable* be ? **IsConcatSpreadable**(*E*).
 - b. If *spreadable* is **true**, then
 - i. Let *k* be 0.
 - ii. Let *len* be ? **LengthOfArrayLike**(*E*).
 - iii. If $n + len > 2^{53} - 1$, throw a **TypeError** exception.
 - iv. Repeat, while $k < len$,
 1. Let *P* be ! **ToString**(**F**(*k*)).
 2. Let *exists* be ? **HasProperty**(*E*, *P*).
 3. If *exists* is **true**, then
 - a. Let *subElement* be ? **Get**(*E*, *P*).
 - b. Perform ? **CreateDataPropertyOrThrow**(*A*, ! **ToString**(**F**(*n*)), *subElement*).
 4. Set *n* to $n + 1$.
 5. Set *k* to $k + 1$.
 - c. Else,
 - i. **NOTE**: *E* is added as a single item rather than spread.
 - ii. If $n \geq 2^{53} - 1$, throw a **TypeError** exception.
 - iii. Perform ? **CreateDataPropertyOrThrow**(*A*, ! **ToString**(**F**(*n*)), *E*).
 - iv. Set *n* to $n + 1$.
6. Perform ? **Set**(*A*, **"length"**, **F**(*n*), **true**).
7. Return *A*.

The **"length"** property of the **concat** method is 1_{F} .

NOTE 1 The explicit setting of the "length" property in step 6 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.

23.1.3.1.1 IsConcatSpreadable (*O*)

The abstract operation IsConcatSpreadable takes argument *O*. It performs the following steps when called:

1. If `Type(O)` is not Object, return **false**.
2. Let *spreadable* be ? `Get(O, @@isConcatSpreadable)`.
3. If *spreadable* is not **undefined**, return ! `ToBoolean(spreadable)`.
4. Return ? `IsArray(O)`.

23.1.3.2 Array.prototype.constructor

The initial value of **Array.prototype.constructor** is `%Array%`.

23.1.3.3 Array.prototype.copyWithIn (*target*, *start* [, *end*])

The **copyWithIn** method takes up to three arguments *target*, *start* and *end*.

NOTE 1 The *end* argument is optional with the length of the **this** value 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 ? `ToObject(this value)`.
2. Let *len* be ? `LengthOfArrayLike(O)`.
3. Let *relativeTarget* be ? `ToIntegerOrInfinity(target)`.
4. If *relativeTarget* is $-\infty$, let *to* be 0.
5. Else if *relativeTarget* < 0, let *to* be `max(len + relativeTarget, 0)`.
6. Else, let *to* be `min(relativeTarget, len)`.
7. Let *relativeStart* be ? `ToIntegerOrInfinity(start)`.
8. If *relativeStart* is $-\infty$, let *from* be 0.
9. Else if *relativeStart* < 0, let *from* be `max(len + relativeStart, 0)`.
10. Else, let *from* be `min(relativeStart, len)`.
11. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? `ToIntegerOrInfinity(end)`.
12. If *relativeEnd* is $-\infty$, let *final* be 0.
13. Else if *relativeEnd* < 0, let *final* be `max(len + relativeEnd, 0)`.
14. Else, let *final* be `min(relativeEnd, len)`.
15. Let *count* be `min(final - from, len - to)`.
16. If *from* < *to* and *to* < *from* + *count*, then
 - a. Let *direction* be -1.
 - b. Set *from* to *from* + *count* - 1.

- c. Set *to* to *to* + *count* - 1.
- 17. Else,
 - a. Let *direction* be 1.
- 18. Repeat, while *count* > 0,
 - a. Let *fromKey* be ! ToString(\mathbb{F} (*from*)).
 - b. Let *toKey* be ! ToString(\mathbb{F} (*to*)).
 - c. Let *fromPresent* be ? HasProperty(*O*, *fromKey*).
 - d. If *fromPresent* is **true**, then
 - i. Let *fromVal* be ? Get(*O*, *fromKey*).
 - ii. Perform ? Set(*O*, *toKey*, *fromVal*, **true**).
 - e. Else,
 - i. **Assert**: *fromPresent* is **false**.
 - ii. Perform ? DeletePropertyOrThrow(*O*, *toKey*).
 - f. Set *from* to *from* + *direction*.
 - g. Set *to* to *to* + *direction*.
 - h. Set *count* to *count* - 1.
- 19. Return *O*.

NOTE 2 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.

23.1.3.4 Array.prototype.entries ()

The following steps are taken:

1. Let *O* be ? ToObject(**this** value).
2. Return CreateArrayIterator(*O*, key+value).

23.1.3.5 Array.prototype.every (*callbackfn* [, *thisArg*])

NOTE 1

callbackfn should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. **every** calls *callbackfn* once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **false**. If such an element is found, **every** immediately returns **false**. Otherwise, if *callbackfn* returned **true** for all elements, **every** will return **true**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

every does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **every** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **every** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time **every** visits them; elements that are deleted after the call to **every** begins and before being visited are not visited. **every** acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns **true**.

When the **every** method is called with one or two arguments, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *k* be 0.
5. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).
 - b. Let *kPresent* be ? **HasProperty**(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? **Get**(*O*, *Pk*).
 - ii. Let *testResult* be ! **ToBoolean**(? **Call**(*callbackfn*, *thisArg*, « *kValue*, **F**(*k*), *O* »)).
 - iii. If *testResult* is **false**, return **false**.
 - d. Set *k* to *k* + 1.
6. Return **true**.

NOTE 2

The **every** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

23.1.3.6 Array.prototype.fill (*value* [, *start* [, *end*]])

The **fill** method takes up to three arguments *value*, *start* and *end*.

NOTE 1 The *start* and *end* arguments are optional with default values of 0 and the length of the **this** value. If *start* is negative, it is treated as $length + start$ where *length* is the length of the array. If *end* is negative, it is treated as $length + end$.

The following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. Let *relativeStart* be ? **ToIntegerOrInfinity**(*start*).
4. If *relativeStart* is $-\infty$, let *k* be 0.
5. Else if *relativeStart* < 0, let *k* be **max**($len + relativeStart$, 0).
6. Else, let *k* be **min**(*relativeStart*, *len*).
7. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? **ToIntegerOrInfinity**(*end*).
8. If *relativeEnd* is $-\infty$, let *final* be 0.
9. Else if *relativeEnd* < 0, let *final* be **max**($len + relativeEnd$, 0).
10. Else, let *final* be **min**(*relativeEnd*, *len*).
11. Repeat, while *k* < *final*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).
 - b. Perform ? **Set**(*O*, *Pk*, *value*, **true**).
 - c. Set *k* to *k* + 1.
12. Return *O*.

NOTE 2 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.

23.1.3.7 Array.prototype.filter (*callbackfn* [, *thisArg*])

NOTE 1 *callbackfn* should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. **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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *A* be ? **ArraySpeciesCreate**(*O*, 0).
5. Let *k* be 0.
6. Let *to* be 0.
7. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - b. Let *kPresent* be ? **HasProperty**(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? **Get**(*O*, *Pk*).
 - ii. Let *selected* be ! **ToBoolean**(? **Call**(*callbackfn*, *thisArg*, « *kValue*, **F**(*k*), *O* »)).
 - iii. If *selected* is **true**, then
 1. Perform ? **CreateDataPropertyOrThrow**(*A*, ! **Tostring**(**F**(*to*)), *kValue*).
 2. Set *to* to *to* + 1.
 - d. Set *k* to *k* + 1.
8. Return *A*.

NOTE 2

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.

23.1.3.8 Array.prototype.find (*predicate* [, *thisArg*])

The **find** method is called with one or two arguments, *predicate* and *thisArg*.

NOTE 1

predicate should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. **find** calls *predicate* once for each element of 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 *predicate*. Elements that are appended to the array after the call to **find** begins will not be visited by *predicate*. 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, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*predicate*) is **false**, throw a **TypeError** exception.
4. Let *k* be 0.
5. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - b. Let *kValue* be ? **Get**(*O*, *Pk*).
 - c. Let *testResult* be ! **ToBoolean**(? **Call**(*predicate*, *thisArg*, « *kValue*, **F**(*k*), *O* »)).
 - d. If *testResult* is **true**, return *kValue*.
 - e. Set *k* to *k* + 1.
6. Return **undefined**.

NOTE 2 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.

23.1.3.9 Array.prototype.findIndex (*predicate* [, *thisArg*])

NOTE 1 *predicate* should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. **findIndex** calls *predicate* once for each element of 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 *predicate*. Elements that are appended to the array after the call to **findIndex** begins will not be visited by *predicate*. 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*predicate*) is **false**, throw a **TypeError** exception.
4. Let *k* be 0.
5. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - b. Let *kValue* be ? **Get**(*O*, *Pk*).
 - c. Let *testResult* be ! **ToBoolean**(? **Call**(*predicate*, *thisArg*, « *kValue*, **F**(*k*), *O* »)).
 - d. If *testResult* is **true**, return **F**(*k*).

- e. Set k to $k + 1$.
6. Return $-1_{\mathbb{F}}$.

NOTE 2 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.

23.1.3.10 Array.prototype.flat ([*depth*])

When the **flat** method is called with zero or one arguments, the following steps are taken:

1. Let O be ? **ToObject**(**this** value).
2. Let $sourceLen$ be ? **LengthOfArrayLike**(O).
3. Let $depthNum$ be 1.
4. If $depth$ is not **undefined**, then
 - a. Set $depthNum$ to ? **ToIntegerOrInfinity**($depth$).
 - b. If $depthNum < 0$, set $depthNum$ to 0.
5. Let A be ? **ArraySpeciesCreate**(O , 0).
6. Perform ? **FlattenIntoArray**(A , O , $sourceLen$, 0, $depthNum$).
7. Return A .

23.1.3.10.1 FlattenIntoArray (*target*, *source*, *sourceLen*, *start*, *depth* [, *mapperFunction*, *thisArg*])

The abstract operation **FlattenIntoArray** takes arguments $target$, $source$, $sourceLen$ (a non-negative **integer**), $start$ (a non-negative **integer**), and $depth$ (a non-negative **integer** or $+\infty$) and optional arguments $mapperFunction$ and $thisArg$. It performs the following steps when called:

1. **Assert**: **Type**($target$) is Object.
2. **Assert**: **Type**($source$) is Object.
3. **Assert**: If $mapperFunction$ is present, then ! **IsCallable**($mapperFunction$) is **true**, $thisArg$ is present, and $depth$ is 1.
4. Let $targetIndex$ be $start$.
5. Let $sourceIndex$ be $+0_{\mathbb{F}}$.
6. Repeat, while $\mathbb{R}(sourceIndex) < sourceLen$,
 - a. Let P be ! **ToString**($sourceIndex$).
 - b. Let $exists$ be ? **HasProperty**($source$, P).
 - c. If $exists$ is **true**, then
 - i. Let $element$ be ? **Get**($source$, P).
 - ii. If $mapperFunction$ is present, then
 1. Set $element$ to ? **Call**($mapperFunction$, $thisArg$, « $element$, $sourceIndex$, $source$ »).
 - iii. Let $shouldFlatten$ be **false**.
 - iv. If $depth > 0$, then
 1. Set $shouldFlatten$ to ? **isArray**($element$).
 - v. If $shouldFlatten$ is **true**, then
 1. If $depth$ is $+\infty$, let $newDepth$ be $+\infty$.
 2. Else, let $newDepth$ be $depth - 1$.
 3. Let $elementLen$ be ? **LengthOfArrayLike**($element$).
 4. Set $targetIndex$ to ? **FlattenIntoArray**($target$, $element$, $elementLen$, $targetIndex$, $newDepth$).
 - vi. Else,
 1. If $targetIndex \geq 2^{53} - 1$, throw a **TypeError** exception.

2. Perform ? `CreateDataPropertyOrThrow`(*target*, ! `ToString`(`ℱ`(*targetIndex*)), *element*).
3. Set *targetIndex* to *targetIndex* + 1.
- d. Set *sourceIndex* to *sourceIndex* + 1_ℱ.
7. Return *targetIndex*.

23.1.3.11 `Array.prototype.flatMap` (*mapperFunction* [, *thisArg*])

When the **flatMap** method is called with one or two arguments, the following steps are taken:

1. Let *O* be ? `ToObject`(**this** value).
2. Let *sourceLen* be ? `LengthOfArrayLike`(*O*).
3. If ! `IsCallable`(*mapperFunction*) is **false**, throw a **TypeError** exception.
4. Let *A* be ? `ArraySpeciesCreate`(*O*, 0).
5. Perform ? `FlattenIntoArray`(*A*, *O*, *sourceLen*, 0, 1, *mapperFunction*, *thisArg*).
6. Return *A*.

23.1.3.12 `Array.prototype.forEach` (*callbackfn* [, *thisArg*])

NOTE 1

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 *callbackfn* 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 ? `ToObject`(**this** value).
2. Let *len* be ? `LengthOfArrayLike`(*O*).
3. If `IsCallable`(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *k* be 0.
5. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! `ToString`(`ℱ`(*k*)).
 - b. Let *kPresent* be ? `HasProperty`(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? `Get`(*O*, *Pk*).
 - ii. Perform ? `Call`(*callbackfn*, *thisArg*, « *kValue*, `ℱ`(*k*), *O* »).

- d. Set k to $k + 1$.
6. Return **undefined**.

NOTE 2 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.

23.1.3.13 Array.prototype.includes (*searchElement* [, *fromIndex*])

NOTE 1 **includes** compares *searchElement* to the elements of the array, in ascending order, using the **SameValueZero** algorithm, and if found at any position, returns **true**; otherwise, **false** is returned.

The optional second argument *fromIndex* defaults to $+0_{\mathbb{F}}$ (i.e. the whole array is searched). If it is greater than or equal to the length of the array, **false** is returned, i.e. the array will not be searched. If it is less than $+0_{\mathbb{F}}$, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than $+0_{\mathbb{F}}$, the whole array will be searched.

When the **includes** method is called, the following steps are taken:

1. Let O be ? **ToObject**(**this** value).
2. Let len be ? **LengthOfArrayLike**(O).
3. If len is 0, return **false**.
4. Let n be ? **ToIntegerOrInfinity**(*fromIndex*).
5. **Assert**: If *fromIndex* is **undefined**, then n is 0.
6. If n is $+\infty$, return **false**.
7. Else if n is $-\infty$, set n to 0.
8. If $n \geq 0$, then
 - a. Let k be n .
9. Else,
 - a. Let k be $len + n$.
 - b. If $k < 0$, set k to 0.
10. Repeat, while $k < len$,
 - a. Let $elementK$ be ? **Get**(O , ! **ToString**($\mathbb{F}(k)$)).
 - b. If **SameValueZero**(*searchElement*, $elementK$) is **true**, return **true**.
 - c. Set k to $k + 1$.
11. Return **false**.

NOTE 2 The **includes** 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.

NOTE 3 The **includes** method intentionally differs from the similar **indexOf** method in two ways. First, it uses the **SameValueZero** algorithm, instead of **Strict Equality Comparison**, allowing it to detect NaN array elements. Second, it does not skip missing array elements, instead treating them as **undefined**.

23.1.3.14 Array.prototype.indexOf (*searchElement* [, *fromIndex*])

NOTE 1

indexOf compares *searchElement* to the elements of the array, in ascending order, using the **Strict Equality Comparison** algorithm, and if found at one or more indices, returns the smallest such index; otherwise, $-1_{\mathbb{F}}$ is returned.

The optional second argument *fromIndex* defaults to $+0_{\mathbb{F}}$ (i.e. the whole array is searched). If it is greater than or equal to the length of the array, $-1_{\mathbb{F}}$ is returned, i.e. the array will not be searched. If it is less than $+0_{\mathbb{F}}$, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than $+0_{\mathbb{F}}$, 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If *len* is 0, return $-1_{\mathbb{F}}$.
4. Let *n* be ? **ToIntegerOrInfinity**(*fromIndex*).
5. **Assert**: If *fromIndex* is **undefined**, then *n* is 0.
6. If *n* is $+\infty$, return $-1_{\mathbb{F}}$.
7. Else if *n* is $-\infty$, set *n* to 0.
8. If $n \geq 0$, then
 - a. Let *k* be *n*.
9. Else,
 - a. Let *k* be *len* + *n*.
 - b. If $k < 0$, set *k* to 0.
10. Repeat, while $k < len$,
 - a. Let *kPresent* be ? **HasProperty**(*O*, ! **ToString**(**F**(*k*))).
 - b. If *kPresent* is **true**, then
 - i. Let *elementK* be ? **Get**(*O*, ! **ToString**(**F**(*k*))).
 - ii. Let *same* be the result of performing **Strict Equality Comparison** *searchElement* === *elementK*.
 - iii. If *same* is **true**, return **F**(*k*).
 - c. Set *k* to *k* + 1.
11. Return $-1_{\mathbb{F}}$.

NOTE 2

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.

23.1.3.15 Array.prototype.join (*separator*)

NOTE 1

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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If *separator* is **undefined**, let *sep* be the single-element String ",".

4. Else, let *sep* be ? `ToString(separator)`.
5. Let *R* be the empty String.
6. Let *k* be 0.
7. Repeat, while *k* < *len*,
 - a. If *k* > 0, set *R* to the **string-concatenation** of *R* and *sep*.
 - b. Let *element* be ? `Get(O, ! ToString(F(k)))`.
 - c. If *element* is **undefined** or **null**, let *next* be the empty String; otherwise, let *next* be ? `ToString(element)`.
 - d. Set *R* to the **string-concatenation** of *R* and *next*.
 - e. Set *k* to *k* + 1.
8. Return *R*.

NOTE 2 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.

23.1.3.16 Array.prototype.keys ()

The following steps are taken:

1. Let *O* be ? `ToObject(this value)`.
2. Return `CreateArrayIterator(O, key)`.

23.1.3.17 Array.prototype.lastIndexOf (*searchElement* [, *fromIndex*])

NOTE 1 **lastIndexOf** compares *searchElement* to the elements of the array in descending order using the **Strict Equality Comparison** algorithm, and if found at one or more indices, returns the largest such index; 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 less than `+0`, 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 ? `ToObject(this value)`.
2. Let *len* be ? `LengthOfArrayLike(O)`.
3. If *len* is 0, return `-1`.
4. If *fromIndex* is present, let *n* be ? `ToIntegerOrInfinity(fromIndex)`; else let *n* be *len* - 1.
5. If *n* is `-∞`, return `-1`.
6. If *n* ≥ 0, then
 - a. Let *k* be `min(n, len - 1)`.
7. Else,
 - a. Let *k* be *len* + *n*.
8. Repeat, while *k* ≥ 0,
 - a. Let *kPresent* be ? `HasProperty(O, ! ToString(F(k)))`.
 - b. If *kPresent* is **true**, then
 - i. Let *elementK* be ? `Get(O, ! ToString(F(k)))`.

- ii. Let *same* be the result of performing [Strict Equality Comparison](#) `searchElement === elementK`.
 - iii. If *same* is **true**, return $\mathbb{F}(k)$.
 - c. Set *k* to *k* - 1.
9. Return **-1**.

NOTE 2 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.

23.1.3.18 Array.prototype.map (*callbackfn* [, *thisArg*])

NOTE 1 *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 ? [ToObject](#)(**this** value).
2. Let *len* be ? [LengthOfArrayLike](#)(*O*).
3. If [IsCallable](#)(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *A* be ? [ArraySpeciesCreate](#)(*O*, *len*).
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! [ToString](#)($\mathbb{F}(k)$).
 - b. Let *kPresent* be ? [HasProperty](#)(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? [Get](#)(*O*, *Pk*).
 - ii. Let *mappedValue* be ? [Call](#)(*callbackfn*, *thisArg*, « *kValue*, $\mathbb{F}(k)$, *O* »).
 - iii. Perform ? [CreateDataPropertyOrThrow](#)(*A*, *Pk*, *mappedValue*).
 - d. Set *k* to *k* + 1.
7. Return *A*.

NOTE 2 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.

23.1.3.19 Array.prototype.pop ()

NOTE 1 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If *len* = 0, then
 - a. Perform ? **Set**(*O*, "length", +0_F, **true**).
 - b. Return **undefined**.
4. Else,
 - a. **Assert**: *len* > 0.
 - b. Let *newLen* be **F**(*len* - 1).
 - c. Let *index* be ! **ToString**(*newLen*).
 - d. Let *element* be ? **Get**(*O*, *index*).
 - e. Perform ? **DeletePropertyOrThrow**(*O*, *index*).
 - f. Perform ? **Set**(*O*, "length", *newLen*, **true**).
 - g. Return *element*.

NOTE 2 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.

23.1.3.20 Array.prototype.push (...*items*)

NOTE 1 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. Let *argCount* be the number of elements in *items*.
4. If *len* + *argCount* > 2⁵³ - 1, throw a **TypeError** exception.
5. For each element *E* of *items*, do
 - a. Perform ? **Set**(*O*, ! **ToString**(**F**(*len*)), *E*, **true**).
 - b. Set *len* to *len* + 1.
6. Perform ? **Set**(*O*, "length", **F**(*len*), **true**).
7. Return **F**(*len*).

The "length" property of the **push** method is 1_F.

NOTE 2

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.

23.1.3.21 Array.prototype.reduce (*callbackfn* [, *initialValue*])

NOTE 1

callbackfn should be a function that takes four arguments. **reduce** calls the callback, as a function, once for each element after the first element present in the array, in ascending order.

callbackfn is called with four arguments: the *previousValue* (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 supplied 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 supplied, 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
4. If *len* = 0 and *initialValue* is not present, throw a **TypeError** exception.
5. Let *k* be 0.
6. Let *accumulator* be **undefined**.
7. If *initialValue* is present, then
 - a. Set *accumulator* to *initialValue*.
8. Else,
 - a. Let *kPresent* be **false**.
 - b. Repeat, while *kPresent* is **false** and *k* < *len*,
 - i. Let *Pk* be ! **ToString**(**F**(*k*)).
 - ii. Set *kPresent* to ? **HasProperty**(*O*, *Pk*).
 - iii. If *kPresent* is **true**, then
 1. Set *accumulator* to ? **Get**(*O*, *Pk*).
 - iv. Set *k* to *k* + 1.
 - c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).

- b. Let *kPresent* be ? `HasProperty(O, Pk)`.
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? `Get(O, Pk)`.
 - ii. Set *accumulator* to ? `Call(callbackfn, undefined, « accumulator, kValue, F(k), O »)`.
 - d. Set *k* to *k* + 1.
10. Return *accumulator*.

NOTE 2 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.

23.1.3.22 Array.prototype.reduceRight (*callbackfn* [, *initialValue*])

NOTE 1 *callbackfn* should be a function that takes four arguments. **reduceRight** calls the callback, as a function, once for each element after the first element present in the array, in descending order.

callbackfn is called with four arguments: the *previousValue* (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 supplied 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 supplied, 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 ? `ToObject(this value)`.
2. Let *len* be ? `LengthOfArrayLike(O)`.
3. If `IsCallable(callbackfn)` is **false**, throw a **TypeError** exception.
4. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
5. Let *k* be *len* - 1.
6. Let *accumulator* be **undefined**.
7. If *initialValue* is present, then
 - a. Set *accumulator* to *initialValue*.
8. Else,
 - a. Let *kPresent* be **false**.
 - b. Repeat, while *kPresent* is **false** and *k* ≥ 0,
 - i. Let *Pk* be ! `ToString(F(k))`.

- ii. Set *kPresent* to ? `HasProperty(O, Pk)`.
 - iii. If *kPresent* is **true**, then
 - 1. Set *accumulator* to ? `Get(O, Pk)`.
 - iv. Set *k* to *k* - 1.
 - c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while *k* ≥ 0,
- a. Let *Pk* be ! `ToString(F(k))`.
 - b. Let *kPresent* be ? `HasProperty(O, Pk)`.
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? `Get(O, Pk)`.
 - ii. Set *accumulator* to ? `Call(callbackfn, undefined, « accumulator, kValue, F(k), O »)`.
 - d. Set *k* to *k* - 1.
10. Return *accumulator*.

NOTE 2 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.

23.1.3.23 Array.prototype.reverse ()

NOTE 1 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 ? `ToObject(this value)`.
2. Let *len* be ? `LengthOfArrayLike(O)`.
3. Let *middle* be `floor(len / 2)`.
4. Let *lower* be 0.
5. Repeat, while *lower* ≠ *middle*,
 - a. Let *upper* be *len* - *lower* - 1.
 - b. Let *upperP* be ! `ToString(F(upper))`.
 - c. Let *lowerP* be ! `ToString(F(lower))`.
 - d. Let *lowerExists* be ? `HasProperty(O, lowerP)`.
 - e. If *lowerExists* is **true**, then
 - i. Let *lowerValue* be ? `Get(O, lowerP)`.
 - f. Let *upperExists* be ? `HasProperty(O, upperP)`.
 - g. If *upperExists* is **true**, then
 - i. Let *upperValue* be ? `Get(O, upperP)`.
 - h. If *lowerExists* is **true** and *upperExists* is **true**, then
 - i. Perform ? `Set(O, lowerP, upperValue, true)`.
 - ii. Perform ? `Set(O, upperP, lowerValue, true)`.
 - i. Else if *lowerExists* is **false** and *upperExists* is **true**, then
 - i. Perform ? `Set(O, lowerP, upperValue, true)`.
 - ii. Perform ? `DeletePropertyOrThrow(O, upperP)`.
 - j. Else if *lowerExists* is **true** and *upperExists* is **false**, then
 - i. Perform ? `DeletePropertyOrThrow(O, lowerP)`.
 - ii. Perform ? `Set(O, upperP, lowerValue, true)`.

- k. Else,
 - i. **Assert:** *lowerExists* and *upperExists* are both **false**.
 - ii. No action is required.
 - l. Set *lower* to *lower* + 1.
6. Return *O*.

NOTE 2 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.

23.1.3.24 Array.prototype.shift ()

NOTE 1 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If *len* = 0, then
 - a. Perform ? **Set**(*O*, "length", +0_F, **true**).
 - b. Return **undefined**.
4. Let *first* be ? **Get**(*O*, "0").
5. Let *k* be 1.
6. Repeat, while *k* < *len*,
 - a. Let *from* be ! **ToString**(**F**(*k*)).
 - b. Let *to* be ! **ToString**(**F**(*k* - 1)).
 - c. Let *fromPresent* be ? **HasProperty**(*O*, *from*).
 - d. If *fromPresent* is **true**, then
 - i. Let *fromVal* be ? **Get**(*O*, *from*).
 - ii. Perform ? **Set**(*O*, *to*, *fromVal*, **true**).
 - e. Else,
 - i. **Assert:** *fromPresent* is **false**.
 - ii. Perform ? **DeletePropertyOrThrow**(*O*, *to*).
 - f. Set *k* to *k* + 1.
7. Perform ? **DeletePropertyOrThrow**(*O*, ! **ToString**(**F**(*len* - 1))).
8. Perform ? **Set**(*O*, "length", **F**(*len* - 1), **true**).
9. Return *first*.

NOTE 2 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.

23.1.3.25 Array.prototype.slice (*start*, *end*)

NOTE 1 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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. Let *relativeStart* be ? **ToIntegerOrInfinity**(*start*).
4. If *relativeStart* is $-\infty$, let *k* be 0.
5. Else if *relativeStart* < 0, let *k* be $\max(len + relativeStart, 0)$.
6. Else, let *k* be $\min(relativeStart, len)$.
7. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? **ToIntegerOrInfinity**(*end*).
8. If *relativeEnd* is $-\infty$, let *final* be 0.
9. Else if *relativeEnd* < 0, let *final* be $\max(len + relativeEnd, 0)$.
10. Else, let *final* be $\min(relativeEnd, len)$.
11. Let *count* be $\max(final - k, 0)$.
12. Let *A* be ? **ArraySpeciesCreate**(*O*, *count*).
13. Let *n* be 0.
14. Repeat, while *k* < *final*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).
 - b. Let *kPresent* be ? **HasProperty**(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? **Get**(*O*, *Pk*).
 - ii. Perform ? **CreateDataPropertyOrThrow**(*A*, ! **ToString**(**F**(*n*)), *kValue*).
 - d. Set *k* to *k* + 1.
 - e. Set *n* to *n* + 1.
15. Perform ? **Set**(*A*, **"length"**, **F**(*n*), **true**).
16. Return *A*.

NOTE 2 The explicit setting of the **"length"** property of the result Array in step 15 was necessary in previous editions of ECMAScript to ensure that its length was correct in situations where the trailing elements of the result Array were not present. Setting **"length"** became unnecessary starting in ES2015 when the result Array was initialized to its proper length rather than an empty Array but is carried forward to preserve backward compatibility.

NOTE 3 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.

23.1.3.26 Array.prototype.some (*callbackfn* [, *thisArg*])

NOTE 1

callbackfn should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. **some** calls *callbackfn* once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **true**. If such an element is found, **some** immediately returns **true**. Otherwise, **some** returns **false**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

some does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **some** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **some** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time that **some** visits them; elements that are deleted after the call to **some** begins and before being visited are not visited. **some** acts like the "exists" quantifier in mathematics. In particular, for an empty array, it returns **false**.

When the **some** method is called with one or two arguments, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *k* be 0.
5. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).
 - b. Let *kPresent* be ? **HasProperty**(*O*, *Pk*).
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be ? **Get**(*O*, *Pk*).
 - ii. Let *testResult* be ! **ToBoolean**(? **Call**(*callbackfn*, *thisArg*, « *kValue*, **F**(*k*), *O* »)).
 - iii. If *testResult* is **true**, return **true**.
 - d. Set *k* to *k* + 1.
6. Return **false**.

NOTE 2

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.

23.1.3.27 Array.prototype.sort (*comparefn*)

The elements of this array are sorted. The sort must be stable (that is, elements that compare equal must remain in their original order). If *comparefn* is not **undefined**, it should be a function that accepts two arguments *x* and *y* and returns a negative value if *x* < *y*, zero if *x* = *y*, or a positive value if *x* > *y*.

The following steps are taken:

1. If *comparefn* is not **undefined** and `IsCallable(comparefn)` is **false**, throw a **TypeError** exception.
2. Let *obj* be `? ToObject(this value)`.
3. Let *len* be `? LengthOfArrayLike(obj)`.
4. Let *items* be a new empty **List**.
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *Pk* be `! ToString(F(k))`.
 - b. Let *kPresent* be `? HasProperty(obj, Pk)`.
 - c. If *kPresent* is **true**, then
 - i. Let *kValue* be `? Get(obj, Pk)`.
 - ii. Append *kValue* to *items*.
 - d. Set *k* to *k* + 1.
7. Let *itemCount* be the number of elements in *items*.
8. Sort *items* using an **implementation-defined** sequence of calls to `SortCompare`. If any such call returns an **abrupt completion**, stop before performing any further calls to `SortCompare` or steps in this algorithm and return that completion.
9. Let *j* be 0.
10. Repeat, while *j* < *itemCount*,
 - a. Perform `? Set(obj, ! ToString(F(j)), items[j], true)`.
 - b. Set *j* to *j* + 1.
11. Repeat, while *j* < *len*,
 - a. Perform `? DeletePropertyOrThrow(obj, ! ToString(F(j)))`.
 - b. Set *j* to *j* + 1.
12. Return *obj*.

The *sort order* is the ordering, after completion of this function, of the **integer-indexed** property values of *obj* whose **integer** indexes are less than *len*. The result of the **sort** function is then determined as follows:

The sort order is **implementation-defined** if any of the following conditions is true:

- If *comparefn* is not **undefined** and is not a consistent comparison function for the elements of *items* (see below).
- If *comparefn* is **undefined** and `SortCompare` does not act as a consistent comparison function.
- If *comparefn* is **undefined** and all applications of `ToString`, to any specific value passed as an argument to `SortCompare`, do not produce the same result.

Unless the sort order is specified above to be **implementation-defined**, *items* must satisfy all of the following conditions after executing step 8 of the algorithm above:

- There must be some mathematical permutation π of the non-negative integers less than *itemCount*, such that for every non-negative **integer** *j* less than *itemCount*, the element `old[j]` is exactly the same as `new[$\pi(j)$]`.
- Then for all non-negative integers *j* and *k*, each less than *itemCount*, if `SortCompare(old[j], old[k]) < 0` (see `SortCompare` below), then $\pi(j) < \pi(k)$.

Here the notation `old[j]` is used to refer to *items[j]* before step 8 is executed, and the notation `new[j]` to refer to *items[j]* after step 8 has been executed.

A function *comparefn* is a consistent comparison function for a set of values *S* if all of the requirements below are met for all values *a*, *b*, and *c* (possibly the same value) in the set *S*: The notation $a <_{CF} b$ means `comparefn(a, b) < 0`; $a =_{CF} b$ means `comparefn(a, b) = 0` (of either sign); and $a >_{CF} b$ means `comparefn(a, b) > 0`.

- Calling `comparefn(a, b)` always returns the same value `v` when given a specific pair of values `a` and `b` as its two arguments. Furthermore, `Type(v)` is `Number`, and `v` is not `NaN`. Note that this implies that exactly one of `a <CF b`, `a =CF b`, and `a >CF b` will be true for a given pair of `a` and `b`.
- Calling `comparefn(a, b)` does not modify `obj` or any object on `obj`'s prototype chain.
- `a =CF a` (reflexivity)
- If `a =CF b`, then `b =CF a` (symmetry)
- If `a =CF b` and `b =CF c`, then `a =CF c` (transitivity of `=CF`)
- If `a <CF b` and `b <CF c`, then `a <CF c` (transitivity of `<CF`)
- If `a >CF b` and `b >CF c`, then `a >CF c` (transitivity of `>CF`)

NOTE 1 The above conditions are necessary and sufficient to ensure that `comparefn` divides the set `S` into equivalence classes and that these equivalence classes are totally ordered.

NOTE 2 The `sort` function is intentionally generic; it does not require that its `this` value be an `Array` object. Therefore, it can be transferred to other kinds of objects for use as a method.

23.1.3.27.1 SortCompare (`x`, `y`)

The abstract operation `SortCompare` takes arguments `x` and `y`. It also has access to the `comparefn` argument passed to the current invocation of the `sort` method. It performs the following steps when called:

1. If `x` and `y` are both `undefined`, return `+0F`.
2. If `x` is `undefined`, return `1F`.
3. If `y` is `undefined`, return `-1F`.
4. If `comparefn` is not `undefined`, then
 - a. Let `v` be `? ToNumber(? Call(comparefn, undefined, « x, y »))`.
 - b. If `v` is `NaN`, return `+0F`.
 - c. Return `v`.
5. Let `xString` be `? ToString(x)`.
6. Let `yString` be `? ToString(y)`.
7. Let `xSmaller` be the result of performing `Abstract Relational Comparison xString < yString`.
8. If `xSmaller` is `true`, return `-1F`.
9. Let `ySmaller` be the result of performing `Abstract Relational Comparison yString < xString`.
10. If `ySmaller` is `true`, return `1F`.
11. Return `+0F`.

NOTE 1 Because non-existent property values always compare greater than `undefined` property values, and `undefined` always compares greater than any other value, `undefined` property values always sort to the end of the result, followed by non-existent property values.

NOTE 2 Method calls performed by the `ToString` abstract operations in steps 5 and 6 have the potential to cause `SortCompare` to not behave as a consistent comparison function.

23.1.3.28 Array.prototype.splice (`start`, `deleteCount`, ...`items`)

NOTE 1

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 elements of *items*. An Array object containing the deleted elements (if any) is returned.

The following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*O*).
3. Let *relativeStart* be ? **ToIntegerOrInfinity**(*start*).
4. If *relativeStart* is $-\infty$, let *actualStart* be 0.
5. Else if *relativeStart* < 0, let *actualStart* be **max**(*len* + *relativeStart*, 0).
6. Else, let *actualStart* be **min**(*relativeStart*, *len*).
7. If *start* is not present, then
 - a. Let *insertCount* be 0.
 - b. Let *actualDeleteCount* be 0.
8. Else if *deleteCount* is not present, then
 - a. Let *insertCount* be 0.
 - b. Let *actualDeleteCount* be *len* - *actualStart*.
9. Else,
 - a. Let *insertCount* be the number of elements in *items*.
 - b. Let *dc* be ? **ToIntegerOrInfinity**(*deleteCount*).
 - c. Let *actualDeleteCount* be the result of **clamping** *dc* between 0 and *len* - *actualStart*.
10. If *len* + *insertCount* - *actualDeleteCount* > 2⁵³ - 1, throw a **TypeError** exception.
11. Let *A* be ? **ArraySpeciesCreate**(*O*, *actualDeleteCount*).
12. Let *k* be 0.
13. Repeat, while *k* < *actualDeleteCount*,
 - a. Let *from* be ! **ToString**(**F**(*actualStart* + *k*)).
 - b. Let *fromPresent* be ? **HasProperty**(*O*, *from*).
 - c. If *fromPresent* is **true**, then
 - i. Let *fromValue* be ? **Get**(*O*, *from*).
 - ii. Perform ? **CreateDataPropertyOrThrow**(*A*, ! **ToString**(**F**(*k*)), *fromValue*).
 - d. Set *k* to *k* + 1.
14. Perform ? **Set**(*A*, "**length**", **F**(*actualDeleteCount*), **true**).
15. Let *itemCount* be the number of elements in *items*.
16. If *itemCount* < *actualDeleteCount*, then
 - a. Set *k* to *actualStart*.
 - b. Repeat, while *k* < (*len* - *actualDeleteCount*),
 - i. Let *from* be ! **ToString**(**F**(*k* + *actualDeleteCount*)).
 - ii. Let *to* be ! **ToString**(**F**(*k* + *itemCount*)).
 - iii. Let *fromPresent* be ? **HasProperty**(*O*, *from*).
 - iv. If *fromPresent* is **true**, then
 1. Let *fromValue* be ? **Get**(*O*, *from*).
 2. Perform ? **Set**(*O*, *to*, *fromValue*, **true**).
 - v. Else,
 1. **Assert**: *fromPresent* is **false**.
 2. Perform ? **DeletePropertyOrThrow**(*O*, *to*).
 - vi. Set *k* to *k* + 1.

- c. Set *k* to *len*.
 - d. Repeat, while *k* > (*len* - *actualDeleteCount* + *itemCount*),
 - i. Perform ? `DeletePropertyOrThrow(O, ! ToString(ℱ(k - 1)))`.
 - ii. Set *k* to *k* - 1.
17. Else if *itemCount* > *actualDeleteCount*, then
- a. Set *k* to (*len* - *actualDeleteCount*).
 - b. Repeat, while *k* > *actualStart*,
 - i. Let *from* be ! `ToString(ℱ(k + actualDeleteCount - 1))`.
 - ii. Let *to* be ! `ToString(ℱ(k + itemCount - 1))`.
 - iii. Let *fromPresent* be ? `HasProperty(O, from)`.
 - iv. If *fromPresent* is **true**, then
 - 1. Let *fromValue* be ? `Get(O, from)`.
 - 2. Perform ? `Set(O, to, fromValue, true)`.
 - v. Else,
 - 1. **Assert:** *fromPresent* is **false**.
 - 2. Perform ? `DeletePropertyOrThrow(O, to)`.
 - vi. Set *k* to *k* - 1.
18. Set *k* to *actualStart*.
19. For each element *E* of *items*, do
- a. Perform ? `Set(O, ! ToString(ℱ(k)), E, true)`.
 - b. Set *k* to *k* + 1.
20. Perform ? `Set(O, "length", ℱ(len - actualDeleteCount + itemCount), true)`.
21. Return *A*.

NOTE 2 The explicit setting of the "length" property of the result Array in step 20 was necessary in previous editions of ECMAScript to ensure that its length was correct in situations where the trailing elements of the result Array were not present. Setting "length" became unnecessary starting in ES2015 when the result Array was initialized to its proper length rather than an empty Array but is carried forward to preserve backward compatibility.

NOTE 3 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.

23.1.3.29 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.

NOTE 1 The first edition of ECMA-402 did not include a replacement specification for the **Array.prototype.toLocaleString** method.

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 ? **ToObject**(**this** value).
2. Let *len* be ? **LengthOfArrayLike**(*array*).
3. 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).
4. Let *R* be the empty String.
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. If *k* > 0, then
 - i. Set *R* to the **string-concatenation** of *R* and *separator*.
 - b. Let *nextElement* be ? **Get**(*array*, ! **ToString**(**F**(*k*))).
 - c. If *nextElement* is not **undefined** or **null**, then
 - i. Let *S* be ? **ToString**(? **Invoke**(*nextElement*, "**toLocaleString**").
 - ii. Set *R* to the **string-concatenation** of *R* and *S*.
 - d. Set *k* to *k* + 1.
7. Return *R*.

NOTE 2 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 3 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.

23.1.3.30 Array.prototype.toString ()

When the **toString** method is called, the following steps are taken:

1. Let *array* be ? **ToObject**(**this** value).
2. Let *func* be ? **Get**(*array*, "**join**").
3. If **IsCallable**(*func*) is **false**, set *func* to the intrinsic function `%Object.prototype.toString%`.
4. Return ? **Call**(*func*, *array*).

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.

23.1.3.31 Array.prototype.unshift (...items)

NOTE 1 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 ? **ToObject**(**this** value).
2. Let len be ? **LengthOfArrayLike**(O).
3. Let $argCount$ be the number of elements in $items$.
4. If $argCount > 0$, then
 - a. If $len + argCount > 2^{53} - 1$, throw a **TypeError** exception.
 - b. Let k be len .
 - c. Repeat, while $k > 0$,
 - i. Let $from$ be ! **ToString**($\mathbb{F}(k - 1)$).
 - ii. Let to be ! **ToString**($\mathbb{F}(k + argCount - 1)$).
 - iii. Let $fromPresent$ be ? **HasProperty**($O, from$).
 - iv. If $fromPresent$ is **true**, then
 1. Let $fromValue$ be ? **Get**($O, from$).
 2. Perform ? **Set**($O, to, fromValue, true$).
 - v. Else,
 1. **Assert**: $fromPresent$ is **false**.
 2. Perform ? **DeletePropertyOrThrow**(O, to).
 - vi. Set k to $k - 1$.
 - d. Let j be $+0_{\mathbb{F}}$.
 - e. For each element E of $items$, do
 - i. Perform ? **Set**($O, !\text{ToString}(j), E, true$).
 - ii. Set j to $j + 1_{\mathbb{F}}$.
5. Perform ? **Set**($O, "length", \mathbb{F}(len + argCount), true$).
6. Return $\mathbb{F}(len + argCount)$.

The "length" property of the **unshift** method is $1_{\mathbb{F}}$.

NOTE 2 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.

23.1.3.32 Array.prototype.values ()

The following steps are taken:

1. Let O be ? **ToObject**(**this** value).
2. Return **CreateArrayIterator**($O, value$).

23.1.3.33 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.

23.1.3.34 Array.prototype [@@unscopables]

The initial value of the **@@unscopables** data property is an object created by the following steps:

1. Let $unscopableList$ be ! **OrdinaryObjectCreate**(**null**).
2. Perform ! **CreateDataPropertyOrThrow**($unscopableList, "copyWithin", true$).
3. Perform ! **CreateDataPropertyOrThrow**($unscopableList, "entries", true$).

4. Perform ! `CreateDataPropertyOrThrow(unscopableList, "fill", true)`.
5. Perform ! `CreateDataPropertyOrThrow(unscopableList, "find", true)`.
6. Perform ! `CreateDataPropertyOrThrow(unscopableList, "findIndex", true)`.
7. Perform ! `CreateDataPropertyOrThrow(unscopableList, "flat", true)`.
8. Perform ! `CreateDataPropertyOrThrow(unscopableList, "flatMap", true)`.
9. Perform ! `CreateDataPropertyOrThrow(unscopableList, "includes", true)`.
10. Perform ! `CreateDataPropertyOrThrow(unscopableList, "keys", true)`.
11. Perform ! `CreateDataPropertyOrThrow(unscopableList, "values", true)`.
12. Return *unscopableList*.

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 ECMAScript 2015 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.

23.1.4 Properties of Array Instances

Array instances are Array exotic objects and have the internal methods specified for such objects. Array instances inherit properties from the [Array prototype object](#).

Array instances have a "**length**" property, and a set of enumerable properties with [array index](#) names.

23.1.4.1 length

The "**length**" property of an Array instance is a [data property](#) whose value is always numerically greater than the name of every configurable own property whose name is an [array index](#).

The "**length**" property initially has the attributes { `[[Writable]]: true`, `[[Enumerable]]: false`, `[[Configurable]]: false` }.

NOTE

Reducing the value of the "**length**" property has the side-effect of deleting own array elements whose [array index](#) is between the old and new length values. However, non-configurable properties can not be deleted. Attempting to set the "**length**" property of an Array object to a value that is numerically less than or equal to the largest numeric own [property name](#) of an existing non-configurable [array-indexed](#) property of the array will result in the length being set to a numeric value that is one greater than that non-configurable numeric own [property name](#). See [10.4.2.1](#).

23.1.5 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.

23.1.5.1 CreateArrayIterator (*array*, *kind*)

The abstract operation `CreateArrayIterator` takes arguments *array* and *kind*. This operation is used to create iterator objects for Array methods that return such iterators. It performs the following steps when called:

1. **Assert:** `Type(array)` is Object.
2. **Assert:** *kind* is key+value, key, or value.
3. Let *closure* be a new **Abstract Closure** with no parameters that captures *kind* and *array* and performs the following steps when called:
 - a. Let *index* be 0.
 - b. Repeat,
 - i. If *array* has a `[[TypedArrayName]]` internal slot, then
 1. If `IsDetachedBuffer(array.[[ViewedArrayBuffer]])` is **true**, throw a **TypeError** exception.
 2. Let *len* be `array.[[ArrayLength]]`.
 - ii. Else,
 1. Let *len* be `? LengthOfArrayLike(array)`.
 - iii. If *index* \geq *len*, return **undefined**.
 - iv. If *kind* is key, perform `? Yield(F(index))`.
 - v. Else,
 1. Let *elementKey* be `! ToString(F(index))`.
 2. Let *elementValue* be `? Get(array, elementKey)`.
 3. If *kind* is value, perform `? Yield(elementValue)`.
 4. Else,
 - a. **Assert:** *kind* is key+value.
 - b. Perform `? Yield(! CreateArrayFromList(« F(index), elementValue »))`.
 - vi. Set *index* to *index* + 1.
 4. Return `! CreateIteratorFromClosure(closure, "%ArrayIteratorPrototype%", %ArrayIteratorPrototype%)`.

23.1.5.2 The `%ArrayIteratorPrototype%` Object

The `%ArrayIteratorPrototype%` object:

- has properties that are inherited by all Array Iterator Objects.
- is an **ordinary object**.
- has a `[[Prototype]]` internal slot whose value is `%IteratorPrototype%`.
- has the following properties:

23.1.5.2.1 `%ArrayIteratorPrototype%.next()`

1. Return `? GeneratorResume(this value, empty, "%ArrayIteratorPrototype%")`.

23.1.5.2.2 `%ArrayIteratorPrototype%` `[@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value `"Array Iterator"`.

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

23.2 TypedArray Objects

TypedArray objects present an array-like view of an underlying binary data buffer (25.1). A *TypedArray element type* is

the underlying binary scalar data type that all elements of a *TypedArray* instance have. There is a distinct *TypedArray* constructor, listed in Table 60, for each of the supported element types. Each constructor in Table 60 has a corresponding distinct prototype object.

Table 60: The TypedArray Constructors

Constructor Name and Intrinsic	Element Type	Element Size	Conversion Operation	Description
Int8Array <i>%Int8Array%</i>	Int8	1	ToInt8	8-bit two's complement signed integer
Uint8Array <i>%Uint8Array%</i>	Uint8	1	ToUint8	8-bit unsigned integer
Uint8ClampedArray <i>%Uint8ClampedArray%</i>	Uint8C	1	ToUint8Clamp	8-bit unsigned integer (clamped conversion)
Int16Array <i>%Int16Array%</i>	Int16	2	ToInt16	16-bit two's complement signed integer
Uint16Array <i>%Uint16Array%</i>	Uint16	2	ToUint16	16-bit unsigned integer
Int32Array <i>%Int32Array%</i>	Int32	4	ToInt32	32-bit two's complement signed integer
Uint32Array <i>%Uint32Array%</i>	Uint32	4	ToUint32	32-bit unsigned integer
BigInt64Array <i>%BigInt64Array%</i>	BigInt64	8	ToBigInt64	64-bit two's complement signed integer
BigUint64Array <i>%BigUint64Array%</i>	BigUint64	8	ToBigUint64	64-bit unsigned integer
Float32Array <i>%Float32Array%</i>	Float32	4		32-bit IEEE floating point
Float64Array <i>%Float64Array%</i>	Float64	8		64-bit IEEE floating point

In the definitions below, references to *TypedArray* should be replaced with the appropriate constructor name from the above table.

23.2.1 The *%TypedArray%* Intrinsic Object

The *%TypedArray%* intrinsic object:

- is a constructor function object that all of the *TypedArray* constructor objects inherit from.
- along with its corresponding prototype object, provides common properties that are inherited by all *TypedArray*

constructors and their instances.

- does not have a global name or appear as a property of the [global object](#).
- acts as the abstract superclass of the various *TypedArray* constructors.
- will throw an error when invoked, because it is an abstract class [constructor](#). The *TypedArray* constructors do not perform a **super** call to it.

23.2.1.1 %TypedArray% ()

The *%TypedArray%* constructor performs the following steps:

1. Throw a **TypeError** exception.

The **"length"** property of the *%TypedArray%* constructor function is **+0**.

23.2.2 Properties of the %TypedArray% Intrinsic Object

The *%TypedArray%* intrinsic object:

- has a **[[Prototype]]** internal slot whose value is *%Function.prototype%*.
- has a **"name"** property whose value is **"TypedArray"**.
- has the following properties:

23.2.2.1 %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**, throw a **TypeError** exception.
3. If *mapfn* is **undefined**, let *mapping* be **false**.
4. Else,
 - a. If **IsCallable**(*mapfn*) is **false**, throw a **TypeError** exception.
 - b. Let *mapping* be **true**.
5. Let *usingIterator* be ? **GetMethod**(*source*, **@@iterator**).
6. If *usingIterator* is not **undefined**, then
 - a. Let *values* be ? **IterableToList**(*source*, *usingIterator*).
 - b. Let *len* be the number of elements in *values*.
 - c. Let *targetObj* be ? **TypedArrayCreate**(*C*, « **ℱ**(*len*) »).
 - d. Let *k* be 0.
 - e. Repeat, while *k* < *len*,
 - i. Let *Pk* be ! **ToString**(**ℱ**(*k*)).
 - ii. Let *kValue* be the first element of *values* and remove that element from *values*.
 - iii. If *mapping* is **true**, then
 1. Let *mappedValue* be ? **Call**(*mapfn*, *thisArg*, « *kValue*, **ℱ**(*k*) »).
 - iv. Else, let *mappedValue* be *kValue*.
 - v. Perform ? **Set**(*targetObj*, *Pk*, *mappedValue*, **true**).
 - vi. Set *k* to *k* + 1.
 - f. **Assert**: *values* is now an empty **List**.
 - g. Return *targetObj*.

7. NOTE: *source* is not an Iterable so assume it is already an array-like object.
8. Let *arrayLike* be ! **ToObject**(*source*).
9. Let *len* be ? **LengthOfArrayLike**(*arrayLike*).
10. Let *targetObj* be ? **TypedArrayCreate**(*C*, « **F**(*len*) »).
11. Let *k* be 0.
12. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! **ToString**(**F**(*k*)).
 - b. Let *kValue* be ? **Get**(*arrayLike*, *Pk*).
 - c. If *mapping* is **true**, then
 - i. Let *mappedValue* be ? **Call**(*mapfn*, *thisArg*, « *kValue*, **F**(*k*) »).
 - d. Else, let *mappedValue* be *kValue*.
 - e. Perform ? **Set**(*targetObj*, *Pk*, *mappedValue*, **true**).
 - f. Set *k* to *k* + 1.
13. Return *targetObj*.

23.2.2.2 %TypedArray%.of (...items)

When the **of** method is called with any number of arguments, the following steps are taken:

1. Let *len* be the number of elements in *items*.
2. Let *C* be the **this** value.
3. If **IsConstructor**(*C*) is **false**, throw a **TypeError** exception.
4. Let *newObj* be ? **TypedArrayCreate**(*C*, « **F**(*len*) »).
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *kValue* be *items*[*k*].
 - b. Let *Pk* be ! **ToString**(**F**(*k*)).
 - c. Perform ? **Set**(*newObj*, *Pk*, *kValue*, **true**).
 - d. Set *k* to *k* + 1.
7. Return *newObj*.

23.2.2.3 %TypedArray%.prototype

The initial value of **%TypedArray%.prototype** is the **%TypedArray%** prototype object.

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

23.2.2.4 get %TypedArray% [@@species]

%TypedArray% [@@species] is an **accessor property** whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the **"name"** property of this function is **"get [Symbol.species]"**.

NOTE

`%TypedArray.prototype%` methods normally use their **this** value's `constructor` to create a derived object. However, a subclass `constructor` may over-ride that default behaviour by redefining its `@@species` property.

23.2.3 Properties of the `%TypedArray%` Prototype Object

The `%TypedArray%` prototype object:

- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is `%TypedArray.prototype%`.
- is an **ordinary object**.
- does not have a `[[ViewedArrayBuffer]]` or any other of the internal slots that are specific to `TypedArray` instance objects.

23.2.3.1 `get %TypedArray%.prototype.buffer`

`%TypedArray%.prototype.buffer` is an **accessor property** whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *O* be the **this** value.
2. Perform ? `RequireInternalSlot`(*O*, `[[TypedArrayName]]`).
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. Return *buffer*.

23.2.3.2 `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. Perform ? `RequireInternalSlot`(*O*, `[[TypedArrayName]]`).
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. If `IsDetachedBuffer`(*buffer*) is **true**, return `+0F`.
6. Let *size* be *O*.`[[ByteLength]]`.
7. Return `F`(*size*).

23.2.3.3 `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. Perform ? `RequireInternalSlot`(*O*, `[[TypedArrayName]]`).
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. If `IsDetachedBuffer`(*buffer*) is **true**, return `+0F`.

6. Let *offset* be *O*.[[ByteOffset]].
7. Return $F(offset)$.

23.2.3.4 %TypedArray%.prototype.constructor

The initial value of `%TypedArray%.prototype.constructor` is the `%TypedArray%` intrinsic object.

23.2.3.5 %TypedArray%.prototype.copyWithin (*target*, *start* [, *end*])

The interpretation and use of the arguments of `%TypedArray%.prototype.copyWithin` are the same as for `Array.prototype.copyWithin` as defined in 23.1.3.3.

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? `ValidateTypedArray(O)`.
3. Let *len* be *O*.[[ArrayLength]].
4. Let *relativeTarget* be ? `ToIntegerOrInfinity(target)`.
5. If *relativeTarget* is $-\infty$, let *to* be 0.
6. Else if *relativeTarget* < 0, let *to* be `max(len + relativeTarget, 0)`.
7. Else, let *to* be `min(relativeTarget, len)`.
8. Let *relativeStart* be ? `ToIntegerOrInfinity(start)`.
9. If *relativeStart* is $-\infty$, let *from* be 0.
10. Else if *relativeStart* < 0, let *from* be `max(len + relativeStart, 0)`.
11. Else, let *from* be `min(relativeStart, len)`.
12. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? `ToIntegerOrInfinity(end)`.
13. If *relativeEnd* is $-\infty$, let *final* be 0.
14. Else if *relativeEnd* < 0, let *final* be `max(len + relativeEnd, 0)`.
15. Else, let *final* be `min(relativeEnd, len)`.
16. Let *count* be `min(final - from, len - to)`.
17. If *count* > 0, then
 - a. NOTE: The copying must be performed in a manner that preserves the bit-level encoding of the source data.
 - b. Let *buffer* be *O*.[[ViewedArrayBuffer]].
 - c. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
 - d. Let *typedArrayName* be the String value of *O*.[[TypedArrayName]].
 - e. Let *elementSize* be the Element Size value specified in Table 60 for *typedArrayName*.
 - f. Let *byteOffset* be *O*.[[ByteOffset]].
 - g. Let *toByteIndex* be $to \times elementSize + byteOffset$.
 - h. Let *fromByteIndex* be $from \times elementSize + byteOffset$.
 - i. Let *countBytes* be $count \times elementSize$.
 - j. If $fromByteIndex < toByteIndex$ and $toByteIndex < fromByteIndex + countBytes$, then
 - i. Let *direction* be -1.
 - ii. Set *fromByteIndex* to $fromByteIndex + countBytes - 1$.
 - iii. Set *toByteIndex* to $toByteIndex + countBytes - 1$.
 - k. Else,
 - i. Let *direction* be 1.
 - l. Repeat, while *countBytes* > 0,

- i. Let *value* be `GetValueFromBuffer(buffer, fromByteIndex, Uint8, true, Unordered)`.
- ii. Perform `SetValueInBuffer(buffer, toByteIndex, Uint8, value, true, Unordered)`.
- iii. Set *fromByteIndex* to *fromByteIndex* + *direction*.
- iv. Set *toByteIndex* to *toByteIndex* + *direction*.
- v. Set *countBytes* to *countBytes* - 1.

18. Return *O*.

23.2.3.6 `%TypedArray%.prototype.entries ()`

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform `? ValidateTypedArray(O)`.
3. Return `CreateArrayIterator(O, key+value)`.

23.2.3.7 `%TypedArray%.prototype.every (callbackfn [, thisArg])`

The interpretation and use of the arguments of `%TypedArray%.prototype.every` are the same as for `Array.prototype.every` as defined in 23.1.3.5.

When the **every** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform `? ValidateTypedArray(O)`.
3. Let *len* be *O*.[[`ArrayLength`]]].
4. If `IsCallable(callbackfn)` is **false**, throw a **TypeError** exception.
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! `ToString(F(k))`.
 - b. Let *kValue* be ! `Get(O, Pk)`.
 - c. Let *testResult* be ! `ToBoolean(? Call(callbackfn, thisArg, « kValue, F(k), O »))`.
 - d. If *testResult* is **false**, return **false**.
 - e. Set *k* to *k* + 1.
7. Return **true**.

This function is not generic. The **this** value must be an object with a [[`TypedArrayName`]] internal slot.

23.2.3.8 `%TypedArray%.prototype.fill (value [, start [, end]])`

The interpretation and use of the arguments of `%TypedArray%.prototype.fill` are the same as for `Array.prototype.fill` as defined in 23.1.3.6.

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform `? ValidateTypedArray(O)`.
3. Let *len* be *O*.[[`ArrayLength`]]].
4. If *O*.[[`ContentType`]] is **BigInt**, set *value* to `? ToBigInt(value)`.
5. Otherwise, set *value* to `? ToNumber(value)`.
6. Let *relativeStart* be `? ToIntegerOrInfinity(start)`.

7. If *relativeStart* is $-\infty$, let *k* be 0.
8. Else if *relativeStart* < 0 , let *k* be $\max(\text{len} + \text{relativeStart}, 0)$.
9. Else, let *k* be $\min(\text{relativeStart}, \text{len})$.
10. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? **ToIntegerOrInfinity**(*end*).
11. If *relativeEnd* is $-\infty$, let *final* be 0.
12. Else if *relativeEnd* < 0 , let *final* be $\max(\text{len} + \text{relativeEnd}, 0)$.
13. Else, let *final* be $\min(\text{relativeEnd}, \text{len})$.
14. If **IsDetachedBuffer**(*O*.[[ViewedArrayBuffer]]) is **true**, throw a **TypeError** exception.
15. Repeat, while *k* $<$ *final*,
 - a. Let *Pk* be ! **ToString**($\mathbb{F}(k)$).
 - b. Perform ! **Set**(*O*, *Pk*, *value*, **true**).
 - c. Set *k* to *k* + 1.
16. Return *O*.

23.2.3.9 %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 23.1.3.7.

When the **filter** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? **ValidateTypedArray**(*O*).
3. Let *len* be *O*.[[ArrayLength]].
4. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
5. Let *kept* be a new empty **List**.
6. Let *k* be 0.
7. Let *captured* be 0.
8. Repeat, while *k* $<$ *len*,
 - a. Let *Pk* be ! **ToString**($\mathbb{F}(k)$).
 - b. Let *kValue* be ! **Get**(*O*, *Pk*).
 - c. Let *selected* be ! **ToBoolean**(? **Call**(*callbackfn*, *thisArg*, $\langle kValue, \mathbb{F}(k), O \rangle$)).
 - d. If *selected* is **true**, then
 - i. Append *kValue* to the end of *kept*.
 - ii. Set *captured* to *captured* + 1.
 - e. Set *k* to *k* + 1.
9. Let *A* be ? **TypedArraySpeciesCreate**(*O*, $\langle \mathbb{F}(\text{captured}) \rangle$).
10. Let *n* be 0.
11. For each element *e* of *kept*, do
 - a. Perform ! **Set**(*A*, ! **ToString**($\mathbb{F}(n)$), *e*, **true**).
 - b. Set *n* to *n* + 1.
12. Return *A*.

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.10 %TypedArray%.prototype.find (*predicate* [, *thisArg*])

The interpretation and use of the arguments of %TypedArray%.**prototype.find** are the same as for **Array.prototype.find** as defined in 23.1.3.8.

When the **find** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? `ValidateTypedArray(O)`.
3. Let *len* be *O*.[`ArrayLength`].
4. If `IsCallable(predicate)` is **false**, throw a **TypeError** exception.
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! `Tostring(F(k))`.
 - b. Let *kValue* be ! `Get(O, Pk)`.
 - c. Let *testResult* be ! `ToBoolean(? Call(predicate, thisArg, « kValue, F(k), O »))`.
 - d. If *testResult* is **true**, return *kValue*.
 - e. Set *k* to *k* + 1.
7. Return **undefined**.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.11 `%TypedArray%.prototype.findIndex (predicate [, thisArg])`

The interpretation and use of the arguments of `%TypedArray%.prototype.findIndex` are the same as for **Array.prototype.findIndex** as defined in 23.1.3.9.

When the **findIndex** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? `ValidateTypedArray(O)`.
3. Let *len* be *O*.[`ArrayLength`].
4. If `IsCallable(predicate)` is **false**, throw a **TypeError** exception.
5. Let *k* be 0.
6. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! `Tostring(F(k))`.
 - b. Let *kValue* be ! `Get(O, Pk)`.
 - c. Let *testResult* be ! `ToBoolean(? Call(predicate, thisArg, « kValue, F(k), O »))`.
 - d. If *testResult* is **true**, return `F(k)`.
 - e. Set *k* to *k* + 1.
7. Return **-1**.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.12 `%TypedArray%.prototype.forEach (callbackfn [, thisArg])`

The interpretation and use of the arguments of `%TypedArray%.prototype.forEach` are the same as for **Array.prototype.forEach** as defined in 23.1.3.12.

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? `ValidateTypedArray(O)`.
3. Let *len* be *O*.[`ArrayLength`].
4. If `IsCallable(callbackfn)` is **false**, throw a **TypeError** exception.

5. Let k be 0.
6. Repeat, while $k < len$,
 - a. Let Pk be ! ToString($F(k)$).
 - b. Let $kValue$ be ! Get(O, Pk).
 - c. Perform ? Call($callbackfn, thisArg, \ll kValue, F(k), O \gg$).
 - d. Set k to $k + 1$.
7. Return **undefined**.

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.13 %TypedArray%.prototype.includes (*searchElement* [, *fromIndex*])

The interpretation and use of the arguments of %TypedArray%.**prototype.includes** are the same as for **Array.prototype.includes** as defined in 23.1.3.13.

When the **includes** method is called with one or two arguments, the following steps are taken:

1. Let O be the **this** value.
2. Perform ? ValidateTypedArray(O).
3. Let len be O .[[ArrayLength]].
4. If len is 0, return **false**.
5. Let n be ? ToIntegerOrInfinity(*fromIndex*).
6. **Assert**: If *fromIndex* is **undefined**, then n is 0.
7. If n is $+\infty$, return **false**.
8. Else if n is $-\infty$, set n to 0.
9. If $n \geq 0$, then
 - a. Let k be n .
10. Else,
 - a. Let k be $len + n$.
 - b. If $k < 0$, set k to 0.
11. Repeat, while $k < len$,
 - a. Let $elementK$ be ! Get($O, ! ToString(F(k))$).
 - b. If SameValueZero(*searchElement*, $elementK$) is **true**, return **true**.
 - c. Set k to $k + 1$.
12. Return **false**.

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.14 %TypedArray%.prototype.indexOf (*searchElement* [, *fromIndex*])

The interpretation and use of the arguments of %TypedArray%.**prototype.indexOf** are the same as for **Array.prototype.indexOf** as defined in 23.1.3.14.

When the **indexOf** method is called with one or two arguments, the following steps are taken:

1. Let O be the **this** value.
2. Perform ? ValidateTypedArray(O).
3. Let len be O .[[ArrayLength]].
4. If len is 0, return **-1**.

5. Let n be ? `ToIntegerOrInfinity`(*fromIndex*).
6. **Assert**: If *fromIndex* is **undefined**, then n is 0.
7. If n is $+\infty$, return $-1_{\mathbb{F}}$.
8. Else if n is $-\infty$, set n to 0.
9. If $n \geq 0$, then
 - a. Let k be n .
10. Else,
 - a. Let k be $len + n$.
 - b. If $k < 0$, set k to 0.
11. Repeat, while $k < len$,
 - a. Let *kPresent* be ! `HasProperty`(O , ! `ToString`($\mathbb{F}(k)$)).
 - b. If *kPresent* is **true**, then
 - i. Let *elementK* be ! `Get`(O , ! `ToString`($\mathbb{F}(k)$)).
 - ii. Let *same* be the result of performing `Strict Equality Comparison` $searchElement === elementK$.
 - iii. If *same* is **true**, return $\mathbb{F}(k)$.
 - c. Set k to $k + 1$.
12. Return $-1_{\mathbb{F}}$.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.15 `%TypedArray%.prototype.join (separator)`

The interpretation and use of the arguments of `%TypedArray%.prototype.join` are the same as for **Array.prototype.join** as defined in 23.1.3.15.

When the **join** method is called with one argument *separator*, the following steps are taken:

1. Let O be the **this** value.
2. Perform ? `ValidateTypedArray`(O).
3. Let *len* be O .`[[ArrayLength]]`.
4. If *separator* is **undefined**, let *sep* be the single-element String `"`,`"`.
5. Else, let *sep* be ? `ToString`(*separator*).
6. Let R be the empty String.
7. Let k be 0.
8. Repeat, while $k < len$,
 - a. If $k > 0$, set R to the `string-concatenation` of R and *sep*.
 - b. Let *element* be ! `Get`(O , ! `ToString`($\mathbb{F}(k)$)).
 - c. If *element* is **undefined**, let *next* be the empty String; otherwise, let *next* be ! `ToString`(*element*).
 - d. Set R to the `string-concatenation` of R and *next*.
 - e. Set k to $k + 1$.
9. Return R .

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.16 `%TypedArray%.prototype.keys ()`

The following steps are taken:

1. Let O be the **this** value.

2. Perform ? [ValidateTypedArray](#)(*O*).
3. Return [CreateArrayIterator](#)(*O*, *key*).

23.2.3.17 %TypedArray%.prototype.lastIndexOf (*searchElement* [, *fromIndex*])

The interpretation and use of the arguments of [%TypedArray%.prototype.lastIndexOf](#) are the same as for [Array.prototype.lastIndexOf](#) as defined in 23.1.3.17.

When the [lastIndexOf](#) method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? [ValidateTypedArray](#)(*O*).
3. Let *len* be *O*.[[[ArrayLength](#)]].
4. If *len* is 0, return $-1_{\mathbb{F}}$.
5. If *fromIndex* is present, let *n* be ? [ToIntegerOrInfinity](#)(*fromIndex*); else let *n* be *len* - 1.
6. If *n* is $-\infty$, return $-1_{\mathbb{F}}$.
7. If $n \geq 0$, then
 - a. Let *k* be [min](#)(*n*, *len* - 1).
8. Else,
 - a. Let *k* be *len* + *n*.
9. Repeat, while $k \geq 0$,
 - a. Let *kPresent* be ! [HasProperty](#)(*O*, ! [ToString](#)($\mathbb{F}(k)$)).
 - b. If *kPresent* is **true**, then
 - i. Let *elementK* be ! [Get](#)(*O*, ! [ToString](#)($\mathbb{F}(k)$)).
 - ii. Let *same* be the result of performing [Strict Equality Comparison](#) *searchElement* === *elementK*.
 - iii. If *same* is **true**, return $\mathbb{F}(k)$.
 - c. Set *k* to *k* - 1.
10. Return $-1_{\mathbb{F}}$.

This function is not generic. The **this** value must be an object with a [[[TypedArrayName](#)]] internal slot.

23.2.3.18 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. Perform ? [RequireInternalSlot](#)(*O*, [[[TypedArrayName](#)]]).
3. **Assert**: *O* has [[[ViewedArrayBuffer](#)]] and [[[ArrayLength](#)]] internal slots.
4. Let *buffer* be *O*.[[[ViewedArrayBuffer](#)]].
5. If [IsDetachedBuffer](#)(*buffer*) is **true**, return $+0_{\mathbb{F}}$.
6. Let *length* be *O*.[[[ArrayLength](#)]].
7. Return $\mathbb{F}(\textit{length})$.

This function is not generic. The **this** value must be an object with a [[[TypedArrayName](#)]] internal slot.

23.2.3.19 %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 23.1.3.18.

When the **map** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? [ValidateTypedArray\(O\)](#).
3. Let *len* be *O*.[[ArrayLength]].
4. If [IsCallable\(callbackfn\)](#) is **false**, throw a **TypeError** exception.
5. Let *A* be ? [TypedArraySpeciesCreate\(O, « F\(len\) »\)](#).
6. Let *k* be 0.
7. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! [ToString\(F\(k\)\)](#).
 - b. Let *kValue* be ! [Get\(O, Pk\)](#).
 - c. Let *mappedValue* be ? [Call\(callbackfn, thisArg, « kValue, F\(k\), O »\)](#).
 - d. Perform ? [Set\(A, Pk, mappedValue, true\)](#).
 - e. Set *k* to *k* + 1.
8. Return *A*.

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.20 %TypedArray%.prototype.reduce (callbackfn [, initialValue])

The interpretation and use of the arguments of [%TypedArray%.prototype.reduce](#) are the same as for **Array.prototype.reduce** as defined in 23.1.3.21.

When the **reduce** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? [ValidateTypedArray\(O\)](#).
3. Let *len* be *O*.[[ArrayLength]].
4. If [IsCallable\(callbackfn\)](#) is **false**, throw a **TypeError** exception.
5. If *len* = 0 and *initialValue* is not present, throw a **TypeError** exception.
6. Let *k* be 0.
7. Let *accumulator* be **undefined**.
8. If *initialValue* is present, then
 - a. Set *accumulator* to *initialValue*.
9. Else,
 - a. Let *Pk* be ! [ToString\(F\(k\)\)](#).
 - b. Set *accumulator* to ! [Get\(O, Pk\)](#).
 - c. Set *k* to *k* + 1.
10. Repeat, while *k* < *len*,
 - a. Let *Pk* be ! [ToString\(F\(k\)\)](#).
 - b. Let *kValue* be ! [Get\(O, Pk\)](#).
 - c. Set *accumulator* to ? [Call\(callbackfn, undefined, « accumulator, kValue, F\(k\), O »\)](#).
 - d. Set *k* to *k* + 1.
11. Return *accumulator*.

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.21 %TypedArray%.prototype.reduceRight (*callbackfn* [, *initialValue*])

The interpretation and use of the arguments of %TypedArray%.prototype.reduceRight are the same as for **Array.prototype.reduceRight** as defined in 23.1.3.22.

When the **reduceRight** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? **ValidateTypedArray**(*O*).
3. Let *len* be *O*.[[**ArrayLength**]].
4. If **IsCallable**(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
6. Let *k* be *len* - 1.
7. Let *accumulator* be **undefined**.
8. If *initialValue* is present, then
 - a. Set *accumulator* to *initialValue*.
9. Else,
 - a. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - b. Set *accumulator* to ! **Get**(*O*, *Pk*).
 - c. Set *k* to *k* - 1.
10. Repeat, while *k* ≥ 0,
 - a. Let *Pk* be ! **Tostring**(**F**(*k*)).
 - b. Let *kValue* be ! **Get**(*O*, *Pk*).
 - c. Set *accumulator* to ? **Call**(*callbackfn*, **undefined**, « *accumulator*, *kValue*, **F**(*k*), *O* »).
 - d. Set *k* to *k* - 1.
11. Return *accumulator*.

This function is not generic. The **this** value must be an object with a [[**TypedArrayName**]] internal slot.

23.2.3.22 %TypedArray%.prototype.reverse ()

The interpretation and use of the arguments of %TypedArray%.prototype.reverse are the same as for **Array.prototype.reverse** as defined in 23.1.3.23.

When the **reverse** method is called, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? **ValidateTypedArray**(*O*).
3. Let *len* be *O*.[[**ArrayLength**]].
4. Let *middle* be **floor**(*len* / 2).
5. Let *lower* be 0.
6. Repeat, while *lower* ≠ *middle*,
 - a. Let *upper* be *len* - *lower* - 1.
 - b. Let *upperP* be ! **Tostring**(**F**(*upper*)).
 - c. Let *lowerP* be ! **Tostring**(**F**(*lower*)).
 - d. Let *lowerValue* be ! **Get**(*O*, *lowerP*).
 - e. Let *upperValue* be ! **Get**(*O*, *upperP*).
 - f. Perform ! **Set**(*O*, *lowerP*, *upperValue*, **true**).
 - g. Perform ! **Set**(*O*, *upperP*, *lowerValue*, **true**).
 - h. Set *lower* to *lower* + 1.

7. Return *O*.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.23 `%TypedArray%.prototype.set (source [, offset])`

`%TypedArray%.prototype.set` is a function whose behaviour differs based upon the type of its first argument.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

Sets multiple values in this *TypedArray*, reading the values from *source*. 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. Let *target* be the **this** value.
2. Perform ? `RequireInternalSlot(target, [[TypedArrayName]])`.
3. **Assert**: *target* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *targetOffset* be ? `ToIntegerOrInfinity(offset)`.
5. If *targetOffset* < 0, throw a **RangeError** exception.
6. If *source* is an Object that has a `[[TypedArrayName]]` internal slot, then
 - a. Perform ? `SetTypedArrayFromTypedArray(target, targetOffset, source)`.
7. Else,
 - a. Perform ? `SetTypedArrayFromArrayLike(target, targetOffset, source)`.
8. Return **undefined**.

23.2.3.23.1 `SetTypedArrayFromTypedArray (target, targetOffset, source)`

The abstract operation `SetTypedArrayFromTypedArray` takes arguments *target* (a *TypedArray* object), *targetOffset* (a non-negative **integer** or $+\infty$), and *source* (a *TypedArray* object). It sets multiple values in *target*, starting at index *targetOffset*, reading the values from *source*. It performs the following steps when called:

1. **Assert**: *source* is an Object that has a `[[TypedArrayName]]` internal slot.
2. Let *targetBuffer* be *target*.`[[ViewedArrayBuffer]]`.
3. If `IsDetachedBuffer(targetBuffer)` is **true**, throw a **TypeError** exception.
4. Let *targetLength* be *target*.`[[ArrayLength]]`.
5. Let *srcBuffer* be *source*.`[[ViewedArrayBuffer]]`.
6. If `IsDetachedBuffer(srcBuffer)` is **true**, throw a **TypeError** exception.
7. Let *targetName* be the String value of *target*.`[[TypedArrayName]]`.
8. Let *targetType* be the Element Type value in Table 60 for *targetName*.
9. Let *targetElementSize* be the Element Size value specified in Table 60 for *targetName*.
10. Let *targetByteOffset* be *target*.`[[ByteOffset]]`.
11. Let *srcName* be the String value of *source*.`[[TypedArrayName]]`.
12. Let *srcType* be the Element Type value in Table 60 for *srcName*.
13. Let *srcElementSize* be the Element Size value specified in Table 60 for *srcName*.
14. Let *srcLength* be *source*.`[[ArrayLength]]`.
15. Let *srcByteOffset* be *source*.`[[ByteOffset]]`.
16. If *targetOffset* is $+\infty$, throw a **RangeError** exception.
17. If *srcLength* + *targetOffset* > *targetLength*, throw a **RangeError** exception.
18. If *target*.`[[ContentType]]` \neq *source*.`[[ContentType]]`, throw a **TypeError** exception.
19. If both `IsSharedArrayBuffer(srcBuffer)` and `IsSharedArrayBuffer(targetBuffer)` are **true**, then
 - a. If *srcBuffer*.`[[ArrayBufferData]]` and *targetBuffer*.`[[ArrayBufferData]]` are the same **Shared Data Block**

- values, let *same* be **true**; else let *same* be **false**.
20. Else, let *same* be `SameValue(srcBuffer, targetBuffer)`.
 21. If *same* is **true**, then
 - a. Let *srcByteLength* be *source*.[[ByteLength]].
 - b. Set *srcBuffer* to ? `CloneArrayBuffer(srcBuffer, srcByteOffset, srcByteLength, %ArrayBuffer%)`.
 - c. NOTE: `%ArrayBuffer%` is used to clone *srcBuffer* because it is known to not have any observable side-effects.
 - d. Let *srcByteIndex* be 0.
 22. Else, let *srcByteIndex* be *srcByteOffset*.
 23. Let *targetByteIndex* be *targetOffset* × *targetElementSize* + *targetByteOffset*.
 24. Let *limit* be *targetByteIndex* + *targetElementSize* × *srcLength*.
 25. If *srcType* is the same as *targetType*, then
 - a. NOTE: If *srcType* and *targetType* are the same, the transfer must be performed in a manner that preserves the bit-level encoding of the source data.
 - b. Repeat, while *targetByteIndex* < *limit*,
 - i. Let *value* be `GetValueFromBuffer(srcBuffer, srcByteIndex, Uint8, true, Unordered)`.
 - ii. Perform `SetValueInBuffer(targetBuffer, targetByteIndex, Uint8, value, true, Unordered)`.
 - iii. Set *srcByteIndex* to *srcByteIndex* + 1.
 - iv. Set *targetByteIndex* to *targetByteIndex* + 1.
 26. Else,
 - a. Repeat, while *targetByteIndex* < *limit*,
 - i. Let *value* be `GetValueFromBuffer(srcBuffer, srcByteIndex, srcType, true, Unordered)`.
 - ii. Perform `SetValueInBuffer(targetBuffer, targetByteIndex, targetType, value, true, Unordered)`.
 - iii. Set *srcByteIndex* to *srcByteIndex* + *srcElementSize*.
 - iv. Set *targetByteIndex* to *targetByteIndex* + *targetElementSize*.

23.2.3.23.2 SetTypedArrayFromArrayLike (*target*, *targetOffset*, *source*)

The abstract operation `SetTypedArrayFromArrayLike` takes arguments *target* (a TypedArray object), *targetOffset* (a non-negative integer or $+\infty$), and *source* (an ECMAScript value other than a TypedArray object). It sets multiple values in *target*, starting at index *targetOffset*, reading the values from *source*. It performs the following steps when called:

1. **Assert**: *source* is any ECMAScript language value other than an Object with a [[TypedArrayName]] internal slot.
2. Let *targetBuffer* be *target*.[[ViewedArrayBuffer]].
3. If `IsDetachedBuffer(targetBuffer)` is **true**, throw a **TypeError** exception.
4. Let *targetLength* be *target*.[[ArrayLength]].
5. Let *targetName* be the String value of *target*.[[TypedArrayName]].
6. Let *targetElementSize* be the Element Size value specified in Table 60 for *targetName*.
7. Let *targetType* be the Element Type value in Table 60 for *targetName*.
8. Let *targetByteOffset* be *target*.[[ByteOffset]].
9. Let *src* be ? `ToObject(source)`.
10. Let *srcLength* be ? `LengthOfArrayLike(src)`.
11. If *targetOffset* is $+\infty$, throw a **RangeError** exception.
12. If *srcLength* + *targetOffset* > *targetLength*, throw a **RangeError** exception.
13. Let *targetByteIndex* be *targetOffset* × *targetElementSize* + *targetByteOffset*.
14. Let *k* be 0.
15. Let *limit* be *targetByteIndex* + *targetElementSize* × *srcLength*.

16. Repeat, while *targetByteIndex* < *limit*,
 - a. Let *Pk* be ! ToString($\mathbb{F}(k)$).
 - b. Let *value* be ? Get(*src*, *Pk*).
 - c. If *target*.[[ContentType]] is BigInt, set *value* to ? ToBigInt(*value*).
 - d. Otherwise, set *value* to ? ToNumber(*value*).
 - e. If IsDetachedBuffer(*targetBuffer*) is true, throw a TypeError exception.
 - f. Perform SetValueInBuffer(*targetBuffer*, *targetByteIndex*, *targetType*, *value*, true, Unordered).
 - g. Set *k* to *k* + 1.
 - h. Set *targetByteIndex* to *targetByteIndex* + *targetElementSize*.

23.2.3.24 %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 23.1.3.25. The following steps are taken:

1. Let *O* be the this value.
2. Perform ? ValidateTypedArray(*O*).
3. Let *len* be *O*.[[ArrayLength]].
4. Let *relativeStart* be ? ToIntegerOrInfinity(*start*).
5. If *relativeStart* is $-\infty$, let *k* be 0.
6. Else if *relativeStart* < 0, let *k* be $\max(\text{len} + \text{relativeStart}, 0)$.
7. Else, let *k* be $\min(\text{relativeStart}, \text{len})$.
8. If *end* is undefined, let *relativeEnd* be *len*; else let *relativeEnd* be ? ToIntegerOrInfinity(*end*).
9. If *relativeEnd* is $-\infty$, let *final* be 0.
10. Else if *relativeEnd* < 0, let *final* be $\max(\text{len} + \text{relativeEnd}, 0)$.
11. Else, let *final* be $\min(\text{relativeEnd}, \text{len})$.
12. Let *count* be $\max(\text{final} - k, 0)$.
13. Let *A* be ? TypedArraySpeciesCreate(*O*, « $\mathbb{F}(\text{count})$ »).
14. If *count* > 0, then
 - a. If IsDetachedBuffer(*O*.[[ViewedArrayBuffer]]) is true, throw a TypeError exception.
 - b. Let *srcName* be the String value of *O*.[[TypedArrayName]].
 - c. Let *srcType* be the Element Type value in Table 60 for *srcName*.
 - d. Let *targetName* be the String value of *A*.[[TypedArrayName]].
 - e. Let *targetType* be the Element Type value in Table 60 for *targetName*.
 - f. If *srcType* is different from *targetType*, then
 - i. Let *n* be 0.
 - ii. Repeat, while *k* < *final*,
 1. Let *Pk* be ! ToString($\mathbb{F}(k)$).
 2. Let *kValue* be ! Get(*O*, *Pk*).
 3. Perform ! Set(*A*, ! ToString($\mathbb{F}(n)$), *kValue*, true).
 4. Set *k* to *k* + 1.
 5. Set *n* to *n* + 1.
 - g. Else,
 - i. Let *srcBuffer* be *O*.[[ViewedArrayBuffer]].
 - ii. Let *targetBuffer* be *A*.[[ViewedArrayBuffer]].
 - iii. Let *elementSize* be the Element Size value specified in Table 60 for Element Type *srcType*.
 - iv. NOTE: If *srcType* and *targetType* are the same, the transfer must be performed in a manner that preserves the bit-level encoding of the source data.

- v. Let *srcByteOffset* be $O.[[\text{ByteOffset}]]$.
 - vi. Let *targetByteIndex* be $A.[[\text{ByteOffset}]]$.
 - vii. Let *srcByteIndex* be $(k \times \text{elementSize}) + \text{srcByteOffset}$.
 - viii. Let *limit* be $\text{targetByteIndex} + \text{count} \times \text{elementSize}$.
 - ix. Repeat, while $\text{targetByteIndex} < \text{limit}$,
 1. Let *value* be `GetValueFromBuffer`(*srcBuffer*, *srcByteIndex*, `Uint8`, `true`, `Unordered`).
 2. Perform `SetValueInBuffer`(*targetBuffer*, *targetByteIndex*, `Uint8`, *value*, `true`, `Unordered`).
 3. Set *srcByteIndex* to $\text{srcByteIndex} + 1$.
 4. Set *targetByteIndex* to $\text{targetByteIndex} + 1$.
15. Return *A*.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.25 `%TypedArray%.prototype.some (callbackfn [, thisArg])`

The interpretation and use of the arguments of `%TypedArray%.prototype.some` are the same as for `Array.prototype.some` as defined in 23.1.3.26.

When the **some** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? `ValidateTypedArray`(*O*).
3. Let *len* be $O.[[\text{ArrayLength}]]$.
4. If `IsCallable`(*callbackfn*) is **false**, throw a **TypeError** exception.
5. Let *k* be 0.
6. Repeat, while $k < \text{len}$,
 - a. Let *Pk* be ! `ToString`($\mathbb{F}(k)$).
 - b. Let *kValue* be ! `Get`(*O*, *Pk*).
 - c. Let *testResult* be ! `ToBoolean`(? `Call`(*callbackfn*, *thisArg*, $\langle kValue, \mathbb{F}(k), O \rangle$)).
 - d. If *testResult* is **true**, return **true**.
 - e. Set *k* to $k + 1$.
7. Return **false**.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

23.2.3.26 `%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 23.1.3.27. 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.

This function is not generic. The **this** value must be an object with a `[[TypedArrayName]]` internal slot.

Upon entry, the following steps are performed to initialize evaluation of the **sort** function. These steps are used instead of steps 1–3 in 23.1.3.27:

1. If *comparefn* is not **undefined** and `IsCallable`(*comparefn*) is **false**, throw a **TypeError** exception.
2. Let *obj* be the **this** value.
3. Let *buffer* be ? `ValidateTypedArray`(*obj*).

4. Let *len* be *obj*.[[ArrayLength]].

The following version of `SortCompare` is used by `%TypedArray%.prototype.sort`. It performs a numeric comparison rather than the string comparison used in 23.1.3.27.

The abstract operation `TypedArraySortCompare` takes arguments *x* and *y*. It also has access to the *comparefn* and *buffer* values of the current invocation of the `sort` method. It performs the following steps when called:

1. **Assert**: Both `Type(x)` and `Type(y)` are `Number` or both are `BigInt`.
2. If *comparefn* is not `undefined`, then
 - a. Let *v* be `? ToNumber(? Call(comparefn, undefined, « x, y »))`.
 - b. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
 - c. If *v* is `NaN`, return `+0F`.
 - d. Return *v*.
3. If *x* and *y* are both `NaN`, return `+0F`.
4. If *x* is `NaN`, return `1F`.
5. If *y* is `NaN`, return `-1F`.
6. If *x* < *y*, return `-1F`.
7. If *x* > *y*, return `1F`.
8. If *x* is `-0F` and *y* is `+0F`, return `-1F`.
9. If *x* is `+0F` and *y* is `-0F`, return `1F`.
10. Return `+0F`.

NOTE 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.

23.2.3.27 `%TypedArray%.prototype.subarray (begin, end)`

Returns a new `TypedArray` object whose element type 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. Perform `? RequireInternalSlot(O, [[TypedArrayName]])`.
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.[[ViewedArrayBuffer]].
5. Let *srcLength* be *O*.[[ArrayLength]].
6. Let *relativeBegin* be `? ToIntegerOrInfinity(begin)`.
7. If *relativeBegin* is `-∞`, let *beginIndex* be 0.
8. Else if *relativeBegin* < 0, let *beginIndex* be `max(srcLength + relativeBegin, 0)`.
9. Else, let *beginIndex* be `min(relativeBegin, srcLength)`.
10. If *end* is `undefined`, let *relativeEnd* be *srcLength*; else let *relativeEnd* be `? ToIntegerOrInfinity(end)`.
11. If *relativeEnd* is `-∞`, let *endIndex* be 0.
12. Else if *relativeEnd* < 0, let *endIndex* be `max(srcLength + relativeEnd, 0)`.
13. Else, let *endIndex* be `min(relativeEnd, srcLength)`.
14. Let *newLength* be `max(endIndex - beginIndex, 0)`.
15. Let *constructorName* be the String value of *O*.[[TypedArrayName]].

16. Let *elementSize* be the Element Size value specified in Table 60 for *constructorName*.
17. Let *srcByteOffset* be *O*.[[ByteOffset]].
18. Let *beginByteOffset* be *srcByteOffset* + *beginIndex* × *elementSize*.
19. Let *argumentsList* be « *buffer*, $\mathbb{F}(\textit{beginByteOffset})$, $\mathbb{F}(\textit{newLength})$ ».
20. Return ? **TypedArraySpeciesCreate**(*O*, *argumentsList*).

This function is not generic. The **this** value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.28 %TypedArray%.prototype.toLocaleString ([*reserved1* [, *reserved2*]])

%TypedArray%.prototype.toLocaleString is a distinct function that implements the same algorithm as **Array.prototype.toLocaleString** as defined in 23.1.3.29 except that the **this** value'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. **ValidateTypedArray** is applied to the **this** value prior to evaluating the algorithm. If its result is an **abrupt completion** that exception is thrown instead of evaluating the algorithm.

NOTE If the ECMAScript implementation includes the ECMA-402 Internationalization API this function is based upon the algorithm for **Array.prototype.toLocaleString** that is in the ECMA-402 specification.

23.2.3.29 %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 23.1.3.30.

23.2.3.30 %TypedArray%.prototype.values ()

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? **ValidateTypedArray**(*O*).
3. Return **CreateArrayIterator**(*O*, value).

23.2.3.31 %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.

23.2.3.32 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`, return **undefined**.
3. If `O` does not have a `[[TypedArrayName]]` internal slot, return **undefined**.
4. Let `name` be `O.[[TypedArrayName]]`.
5. **Assert**: `Type(name)` is `String`.
6. Return `name`.

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

The initial value of the **"name"** property of this function is **"get [Symbol.toStringTag]"**.

23.2.4 Abstract Operations for TypedArray Objects

23.2.4.1 TypedArraySpeciesCreate (*exemplar*, *argumentList*)

The abstract operation `TypedArraySpeciesCreate` takes arguments *exemplar* and *argumentList*. It is used to specify the creation of a new `TypedArray` object using a `constructor` function that is derived from *exemplar*. It performs the following steps when called:

1. **Assert**: *exemplar* is an `Object` that has `[[TypedArrayName]]` and `[[ContentType]]` internal slots.
2. Let *defaultConstructor* be the intrinsic object listed in column one of [Table 60](#) for *exemplar*.`[[TypedArrayName]]`.
3. Let *constructor* be `? SpeciesConstructor(exemplar, defaultConstructor)`.
4. Let *result* be `? TypedArrayCreate(constructor, argumentList)`.
5. **Assert**: *result* has `[[TypedArrayName]]` and `[[ContentType]]` internal slots.
6. If *result*.`[[ContentType]]` \neq *exemplar*.`[[ContentType]]`, throw a **TypeError** exception.
7. Return *result*.

23.2.4.2 TypedArrayCreate (*constructor*, *argumentList*)

The abstract operation `TypedArrayCreate` takes arguments *constructor* and *argumentList*. It is used to specify the creation of a new `TypedArray` object using a `constructor` function. It performs the following steps when called:

1. Let *newTypedArray* be `? Construct(constructor, argumentList)`.
2. Perform `? ValidateTypedArray(newTypedArray)`.
3. If *argumentList* is a `List` of a single `Number`, then
 - a. If *newTypedArray*.`[[ArrayLength]]` $<$ \mathbb{R} (*argumentList*[0]), throw a **TypeError** exception.
4. Return *newTypedArray*.

23.2.4.3 ValidateTypedArray (*O*)

The abstract operation `ValidateTypedArray` takes argument *O*. It performs the following steps when called:

1. Perform `? RequireInternalSlot(O, [[TypedArrayName]])`.
2. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
3. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
4. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
5. Return *buffer*.

23.2.5 The *TypedArray* Constructors

Each *TypedArray* constructor:

- is an intrinsic object that has the structure described below, differing only in the name used as the **constructor** name instead of *TypedArray*, in Table 60.
- is a function whose behaviour differs based upon the number and types of its arguments. The actual behaviour of a call of *TypedArray* depends upon the number and kind of arguments that are passed to it.
- is not intended to be called as a function and will throw an exception when called in that manner.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified *TypedArray* behaviour must include a **super** call to the *TypedArray* constructor to create and initialize the subclass instance with the internal state necessary to support the *%TypedArray%*.**prototype** built-in methods.
- has a "length" property whose value is 3_F.

23.2.5.1 *TypedArray* (...args)

Each *TypedArray* constructor performs the following steps when called:

1. If *NewTarget* is **undefined**, throw a **TypeError** exception.
2. Let *constructorName* be the String value of the **Constructor** Name value specified in Table 60 for this *TypedArray* constructor.
3. Let *proto* be "**%TypedArray.prototype%**".
4. Let *numberOfArgs* be the number of elements in *args*.
5. If *numberOfArgs* = 0, then
 - a. Return ? **AllocateTypedArray**(*constructorName*, *NewTarget*, *proto*, 0).
6. Else,
 - a. Let *firstArgument* be *args*[0].
 - b. If **Type**(*firstArgument*) is Object, then
 - i. Let *O* be ? **AllocateTypedArray**(*constructorName*, *NewTarget*, *proto*).
 - ii. If *firstArgument* has a **[[TypedArrayName]]** internal slot, then
 1. Perform ? **InitializeTypedArrayFromTypedArray**(*O*, *firstArgument*).
 - iii. Else if *firstArgument* has a **[[ArrayBufferData]]** internal slot, then
 1. If *numberOfArgs* > 1, let *byteOffset* be *args*[1]; else let *byteOffset* be **undefined**.
 2. If *numberOfArgs* > 2, let *length* be *args*[2]; else let *length* be **undefined**.
 3. Perform ? **InitializeTypedArrayFromArrayBuffer**(*O*, *firstArgument*, *byteOffset*, *length*).
 - iv. Else,
 1. **Assert**: **Type**(*firstArgument*) is Object and *firstArgument* does not have either a **[[TypedArrayName]]** or a **[[ArrayBufferData]]** internal slot.
 2. Let *usingIterator* be ? **GetMethod**(*firstArgument*, **@@iterator**).
 3. If *usingIterator* is not **undefined**, then
 - a. Let *values* be ? **IterableToList**(*firstArgument*, *usingIterator*).
 - b. Perform ? **InitializeTypedArrayFromList**(*O*, *values*).
 4. Else,
 - a. NOTE: *firstArgument* is not an Iterable so assume it is already an **array-like object**.
 - b. Perform ? **InitializeTypedArrayFromArrayLike**(*O*, *firstArgument*).
 - v. Return *O*.
 - c. Else,
 - i. **Assert**: *firstArgument* is not an Object.
 - ii. Let *elementLength* be ? **ToIndex**(*firstArgument*).

iii. Return ? `AllocateTypedArray`(*constructorName*, *NewTarget*, *proto*, *elementLength*).

23.2.5.1.1 `AllocateTypedArray` (*constructorName*, *newTarget*, *defaultProto* [, *length*])

The abstract operation `AllocateTypedArray` takes arguments *constructorName* (a String which is the name of a TypedArray constructor in Table 60), *newTarget*, and *defaultProto* and optional argument *length* (a non-negative integer). It is used to validate and create an instance of a TypedArray constructor. If the *length* argument is passed, an `ArrayBuffer` of that length is also allocated and associated with the new TypedArray instance. `AllocateTypedArray` provides common semantics that is used by `TypedArray`. It performs the following steps when called:

1. Let *proto* be ? `GetPrototypeFromConstructor`(*newTarget*, *defaultProto*).
2. Let *obj* be ! `IntegerIndexedObjectCreate`(*proto*).
3. **Assert**: *obj*.[[ViewedArrayBuffer]] is **undefined**.
4. Set *obj*.[[TypedArrayName]] to *constructorName*.
5. If *constructorName* is "BigInt64Array" or "BigUint64Array", set *obj*.[[ContentType]] to `BigInt`.
6. Otherwise, set *obj*.[[ContentType]] to `Number`.
7. If *length* is not present, then
 - a. Set *obj*.[[ByteLength]] to 0.
 - b. Set *obj*.[[ByteOffset]] to 0.
 - c. Set *obj*.[[ArrayLength]] to 0.
8. Else,
 - a. Perform ? `AllocateTypedArrayBuffer`(*obj*, *length*).
9. Return *obj*.

23.2.5.1.2 `InitializeTypedArrayFromTypedArray` (*O*, *srcArray*)

The abstract operation `InitializeTypedArrayFromTypedArray` takes arguments *O* (a TypedArray object) and *srcArray* (a TypedArray object). It performs the following steps when called:

1. **Assert**: *O* is an Object that has a [[TypedArrayName]] internal slot.
2. **Assert**: *srcArray* is an Object that has a [[TypedArrayName]] internal slot.
3. Let *srcData* be *srcArray*.[[ViewedArrayBuffer]].
4. If `IsDetachedBuffer`(*srcData*) is **true**, throw a **TypeError** exception.
5. Let *constructorName* be the String value of *O*.[[TypedArrayName]].
6. Let *elementType* be the Element Type value in Table 60 for *constructorName*.
7. Let *elementLength* be *srcArray*.[[ArrayLength]].
8. Let *srcName* be the String value of *srcArray*.[[TypedArrayName]].
9. Let *srcType* be the Element Type value in Table 60 for *srcName*.
10. Let *srcElementSize* be the Element Size value specified in Table 60 for *srcName*.
11. Let *srcByteOffset* be *srcArray*.[[ByteOffset]].
12. Let *elementSize* be the Element Size value specified in Table 60 for *constructorName*.
13. Let *byteLength* be *elementSize* × *elementLength*.
14. If `IsSharedArrayBuffer`(*srcData*) is **false**, then
 - a. Let *bufferConstructor* be ? `SpeciesConstructor`(*srcData*, `%ArrayBuffer%`).
15. Else,
 - a. Let *bufferConstructor* be `%ArrayBuffer%`.
16. If *elementType* is the same as *srcType*, then
 - a. Let *data* be ? `CloneArrayBuffer`(*srcData*, *srcByteOffset*, *byteLength*, *bufferConstructor*).
17. Else,

- a. Let *data* be ? `AllocateArrayBuffer(bufferConstructor, byteLength)`.
- b. If `IsDetachedBuffer(srcData)` is **true**, throw a **TypeError** exception.
- c. If `srcArray.[[ContentType]] ≠ O.[[ContentType]]`, throw a **TypeError** exception.
- d. Let *srcByteIndex* be `srcByteOffset`.
- e. Let *targetByteIndex* be 0.
- f. Let *count* be `elementLength`.
- g. Repeat, while *count* > 0,
 - i. Let *value* be `GetValueFromBuffer(srcData, srcByteIndex, srcType, true, Unordered)`.
 - ii. Perform `SetValueInBuffer(data, targetByteIndex, elementType, value, true, Unordered)`.
 - iii. Set *srcByteIndex* to `srcByteIndex + srcElementSize`.
 - iv. Set *targetByteIndex* to `targetByteIndex + elementSize`.
 - v. Set *count* to *count* - 1.
18. Set `O.[[ViewedArrayBuffer]]` to *data*.
19. Set `O.[[ByteLength]]` to *byteLength*.
20. Set `O.[[ByteOffset]]` to 0.
21. Set `O.[[ArrayLength]]` to *elementLength*.

23.2.5.1.3 InitializeTypedArrayFromArrayBuffer (*O*, *buffer*, *byteOffset*, *length*)

The abstract operation `InitializeTypedArrayFromArrayBuffer` takes arguments *O* (a TypedArray object), *buffer* (an ArrayBuffer object), *byteOffset* (an ECMAScript language value), and *length* (an ECMAScript language value). It performs the following steps when called:

1. **Assert**: *O* is an Object that has a `[[TypedArrayName]]` internal slot.
2. **Assert**: *buffer* is an Object that has an `[[ArrayBufferData]]` internal slot.
3. Let *constructorName* be the String value of `O.[[TypedArrayName]]`.
4. Let *elementSize* be the Element Size value specified in Table 60 for *constructorName*.
5. Let *offset* be ? `ToIndex(byteOffset)`.
6. If *offset* modulo *elementSize* ≠ 0, throw a **RangeError** exception.
7. If *length* is not **undefined**, then
 - a. Let *newLength* be ? `ToIndex(length)`.
8. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
9. Let *bufferByteLength* be `buffer.[[ArrayBufferByteLength]]`.
10. If *length* is **undefined**, then
 - a. If *bufferByteLength* modulo *elementSize* ≠ 0, throw a **RangeError** exception.
 - b. Let *newByteLength* be `bufferByteLength - offset`.
 - c. If *newByteLength* < 0, throw a **RangeError** exception.
11. Else,
 - a. Let *newByteLength* be *newLength* × *elementSize*.
 - b. If *offset* + *newByteLength* > *bufferByteLength*, throw a **RangeError** exception.
12. Set `O.[[ViewedArrayBuffer]]` to *buffer*.
13. Set `O.[[ByteLength]]` to *newByteLength*.
14. Set `O.[[ByteOffset]]` to *offset*.
15. Set `O.[[ArrayLength]]` to *newByteLength* / *elementSize*.

23.2.5.1.4 InitializeTypedArrayFromList (*O*, *values*)

The abstract operation `InitializeTypedArrayFromList` takes arguments *O* (a TypedArray object) and *values* (a List of ECMAScript language values). It performs the following steps when called:

1. **Assert:** O is an Object that has a `[[TypedArrayName]]` internal slot.
2. Let len be the number of elements in $values$.
3. Perform ? `AllocateTypedArrayBuffer`(O, len).
4. Let k be 0.
5. Repeat, while $k < len$,
 - a. Let Pk be ! `ToString`($F(k)$).
 - b. Let $kValue$ be the first element of $values$ and remove that element from $values$.
 - c. Perform ? `Set`($O, Pk, kValue, true$).
 - d. Set k to $k + 1$.
6. **Assert:** $values$ is now an empty List.

23.2.5.1.5 InitializeTypedArrayFromArrayLike ($O, arrayLike$)

The abstract operation `InitializeTypedArrayFromArrayLike` takes arguments O (a `TypedArray` object) and $arrayLike$ (an Object that is neither a `TypedArray` object nor an `ArrayBuffer` object). It performs the following steps when called:

1. **Assert:** O is an Object that has a `[[TypedArrayName]]` internal slot.
2. Let len be ? `LengthOfArrayLike`($arrayLike$).
3. Perform ? `AllocateTypedArrayBuffer`(O, len).
4. Let k be 0.
5. Repeat, while $k < len$,
 - a. Let Pk be ! `ToString`($F(k)$).
 - b. Let $kValue$ be ? `Get`($arrayLike, Pk$).
 - c. Perform ? `Set`($O, Pk, kValue, true$).
 - d. Set k to $k + 1$.

23.2.5.1.6 AllocateTypedArrayBuffer ($O, length$)

The abstract operation `AllocateTypedArrayBuffer` takes arguments O (a `TypedArray` object) and $length$ (a non-negative integer). It allocates and associates an `ArrayBuffer` with O . It performs the following steps when called:

1. **Assert:** O is an Object that has a `[[ViewedArrayBuffer]]` internal slot.
2. **Assert:** O .`[[ViewedArrayBuffer]]` is **undefined**.
3. Let $constructorName$ be the String value of O .`[[TypedArrayName]]`.
4. Let $elementSize$ be the Element Size value specified in Table 60 for $constructorName$.
5. Let $byteLength$ be $elementSize \times length$.
6. Let $data$ be ? `AllocateArrayBuffer`(`%ArrayBuffer%`, $byteLength$).
7. Set O .`[[ViewedArrayBuffer]]` to $data$.
8. Set O .`[[ByteLength]]` to $byteLength$.
9. Set O .`[[ByteOffset]]` to 0.
10. Set O .`[[ArrayLength]]` to $length$.
11. Return O .

23.2.6 Properties of the `TypedArray` Constructors

Each `TypedArray` constructor:

- has a `[[Prototype]]` internal slot whose value is `%TypedArray%`.
- has a **"name"** property whose value is the String value of the `constructor` name specified for it in Table 60.

- has the following properties:

23.2.6.1 *TypedArray*.BYTES_PER_ELEMENT

The value of *TypedArray*.**BYTES_PER_ELEMENT** is the Element Size value specified in Table 60 for *TypedArray*.

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

23.2.6.2 *TypedArray*.prototype

The initial value of *TypedArray*.**prototype** is the corresponding *TypedArray* prototype intrinsic object (23.2.7).

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

23.2.7 Properties of the *TypedArray* Prototype Objects

Each *TypedArray* prototype object:

- has a [[Prototype]] internal slot whose value is %*TypedArray*.prototype%.
- is an [ordinary object](#).
- does not have a [[ViewedArrayBuffer]] or any other of the internal slots that are specific to *TypedArray* instance objects.

23.2.7.1 *TypedArray*.prototype.BYTES_PER_ELEMENT

The value of *TypedArray*.**prototype.BYTES_PER_ELEMENT** is the Element Size value specified in Table 60 for *TypedArray*.

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

23.2.7.2 *TypedArray*.prototype.constructor

The initial value of a *TypedArray*.**prototype.constructor** is the corresponding %*TypedArray*% intrinsic object.

23.2.8 Properties of *TypedArray* Instances

TypedArray instances are [Integer-Indexed exotic objects](#). Each *TypedArray* instance inherits properties from the corresponding *TypedArray* prototype object. Each *TypedArray* instance has the following internal slots: [[TypedArrayName]], [[ViewedArrayBuffer]], [[ByteLength]], [[ByteOffset]], and [[ArrayLength]].

24 Keyed Collections

24.1 Map Objects

Map objects are collections of key/value pairs where both the keys and values may be arbitrary ECMAScript

language values. A distinct key value may only occur in one key/value pair within the Map's collection. Distinct key values 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.

24.1.1 The Map Constructor

The Map **constructor**:

- is `%Map%`.
- is the initial value of the "Map" property of the [global object](#).
- creates and initializes a new Map object when called as a **constructor**.
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 the Map **constructor** to create and initialize the subclass instance with the internal state necessary to support the **Map.prototype** built-in methods.

24.1.1.1 Map ([*iterable*])

When the **Map** function is called with optional argument *iterable*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *map* be ? [OrdinaryCreateFromConstructor](#)(NewTarget, "%Map.prototype%", « [[MapData]] »).
3. Set *map*.[[MapData]] to a new empty [List](#).
4. If *iterable* is either **undefined** or **null**, return *map*.
5. Let *adder* be ? [Get](#)(*map*, "set").
6. Return ? [AddEntriesFromIterable](#)(*map*, *iterable*, *adder*).

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.

24.1.1.2 AddEntriesFromIterable (*target*, *iterable*, *adder*)

The abstract operation AddEntriesFromIterable takes arguments *target*, *iterable*, and *adder* (a [function object](#)). *adder* will be invoked, with *target* as the receiver. It performs the following steps when called:

1. If [IsCallable](#)(*adder*) is **false**, throw a **TypeError** exception.
2. **Assert**: *iterable* is present, and is neither **undefined** nor **null**.
3. Let *iteratorRecord* be ? [GetIterator](#)(*iterable*).
4. Repeat,
 - a. Let *next* be ? [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is **false**, return *target*.

- c. Let *nextItem* be ? `IteratorValue(next)`.
- d. If `Type(nextItem)` is not `Object`, then
 - i. Let *error* be `ThrowCompletion`(a newly created `TypeError` object).
 - ii. Return ? `IteratorClose(iteratorRecord, error)`.
- e. Let *k* be `Get(nextItem, "0")`.
- f. If *k* is an `abrupt completion`, return ? `IteratorClose(iteratorRecord, k)`.
- g. Let *v* be `Get(nextItem, "1")`.
- h. If *v* is an `abrupt completion`, return ? `IteratorClose(iteratorRecord, v)`.
- i. Let *status* be `Call(adder, target, « k.[[Value]], v.[[Value]] »)`.
- j. If *status* is an `abrupt completion`, return ? `IteratorClose(iteratorRecord, status)`.

NOTE

The parameter *iterable* 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.

24.1.2 Properties of the Map Constructor

The `Map` constructor:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

24.1.2.1 Map.prototype

The initial value of **Map.prototype** is the `Map` prototype object.

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

24.1.2.2 get Map [@@species]

Map[@@species] is an `accessor property` whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the `"name"` property of this function is `"get [Symbol.species]"`.

NOTE

Methods that create derived collection objects should call `@@species` to determine the `constructor` to use to create the derived objects. Subclass `constructor` may over-ride `@@species` to change the default `constructor` assignment.

24.1.3 Properties of the Map Prototype Object

The `Map` prototype object:

- is `%Map.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.

- is an **ordinary object**.
- does not have a `[[MapData]]` internal slot.

24.1.3.1 `Map.prototype.clear ()`

The following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? `RequireInternalSlot(M, [[MapData]])`.
3. Let *entries* be the **List** that is *M*.`[[MapData]]`.
4. For each **Record** { `[[Key]]`, `[[Value]]` } *p* of *entries*, do
 - a. Set *p*.`[[Key]]` to **empty**.
 - b. Set *p*.`[[Value]]` to **empty**.
5. Return **undefined**.

NOTE The existing `[[MapData]]` **List** is preserved because there may be existing Map Iterator objects that are suspended midway through iterating over that **List**.

24.1.3.2 `Map.prototype.constructor`

The initial value of `Map.prototype.constructor` is `%Map%`.

24.1.3.3 `Map.prototype.delete (key)`

The following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? `RequireInternalSlot(M, [[MapData]])`.
3. Let *entries* be the **List** that is *M*.`[[MapData]]`.
4. For each **Record** { `[[Key]]`, `[[Value]]` } *p* of *entries*, do
 - 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**.
5. 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.

24.1.3.4 `Map.prototype.entries ()`

The following steps are taken:

1. Let *M* be the **this** value.
2. Return ? `CreateMapIterator(M, key+value)`.

24.1.3.5 Map.prototype.forEach (*callbackfn* [, *thisArg*])

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*M*, [[MapData]]).
3. If [IsCallable](#)(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *entries* be the [List](#) that is *M*.[[MapData]].
5. For each [Record](#) { [[Key]], [[Value]] } *e* of *entries*, do
 - a. If *e*.[[Key]] is not **empty**, then
 - i. Perform ? [Call](#)(*callbackfn*, *thisArg*, « *e*.[[Value]], *e*.[[Key]], *M* »).
6. Return **undefined**.

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*. Each entry of a map's [[MapData]] is only visited once. New keys added after the call to **forEach** begins are visited. A key will be revisited if it is deleted after it has been visited and then re-added before the **forEach** call completes. Keys that are deleted after the call to **forEach** begins and before being visited are not visited unless the key is added again before the **forEach** call completes.

24.1.3.6 Map.prototype.get (*key*)

The following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*M*, [[MapData]]).
3. Let *entries* be the [List](#) that is *M*.[[MapData]].
4. For each [Record](#) { [[Key]], [[Value]] } *p* of *entries*, do
 - a. If *p*.[[Key]] is not **empty** and [SameValueZero](#)(*p*.[[Key]], *key*) is **true**, return *p*.[[Value]].
5. Return **undefined**.

24.1.3.7 Map.prototype.has (*key*)

The following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*M*, [[MapData]]).
3. Let *entries* be the [List](#) that is *M*.[[MapData]].

4. For each **Record** { [[Key]], [[Value]] } *p* of *entries*, do
 - a. If *p*.[[Key]] is not empty and **SameValueZero**(*p*.[[Key]], *key*) is **true**, return **true**.
5. Return **false**.

24.1.3.8 Map.prototype.keys ()

The following steps are taken:

1. Let *M* be the **this** value.
2. Return ? **CreateMapIterator**(*M*, key).

24.1.3.9 Map.prototype.set (*key*, *value*)

The following steps are taken:

1. Let *M* be the **this** value.
2. Perform ? **RequireInternalSlot**(*M*, [[MapData]]).
3. Let *entries* be the **List** that is *M*.[[MapData]].
4. For each **Record** { [[Key]], [[Value]] } *p* of *entries*, do
 - a. If *p*.[[Key]] is not empty and **SameValueZero**(*p*.[[Key]], *key*) is **true**, then
 - i. Set *p*.[[Value]] to *value*.
 - ii. Return *M*.
5. If *key* is $-0_{\mathbb{F}}$, set *key* to $+0_{\mathbb{F}}$.
6. Let *p* be the **Record** { [[Key]]: *key*, [[Value]]: *value* }.
7. Append *p* as the last element of *entries*.
8. Return *M*.

24.1.3.10 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. Perform ? **RequireInternalSlot**(*M*, [[MapData]]).
3. Let *entries* be the **List** that is *M*.[[MapData]].
4. Let *count* be 0.
5. For each **Record** { [[Key]], [[Value]] } *p* of *entries*, do
 - a. If *p*.[[Key]] is not empty, set *count* to *count* + 1.
6. Return **F**(*count*).

24.1.3.11 Map.prototype.values ()

The following steps are taken:

1. Let *M* be the **this** value.
2. Return ? **CreateMapIterator**(*M*, value).

24.1.3.12 Map.prototype [@@iterator] ()

The initial value of the `@@iterator` property is the same [function object](#) as the initial value of the `"entries"` property.

24.1.3.13 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` }.

24.1.4 Properties of Map Instances

Map instances are ordinary objects that inherit properties from the Map prototype. Map instances also have a `[[MapData]]` internal slot.

24.1.5 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.

24.1.5.1 CreateMapIterator (*map*, *kind*)

The abstract operation `CreateMapIterator` takes arguments *map* and *kind*. This operation is used to create iterator objects for Map methods that return such iterators. It performs the following steps when called:

1. **Assert:** *kind* is key+value, key, or value.
2. Perform ? `RequireInternalSlot`(*map*, `[[MapData]]`).
3. Let *closure* be a new [Abstract Closure](#) with no parameters that captures *map* and *kind* and performs the following steps when called:
 - a. Let *entries* be the [List](#) that is *map*.`[[MapData]]`.
 - b. Let *index* be 0.
 - c. Let *numEntries* be the number of elements of *entries*.
 - d. Repeat, while *index* < *numEntries*,
 - i. Let *e* be the [Record](#) { `[[Key]]`, `[[Value]]` } that is the value of *entries*[*index*].
 - ii. Set *index* to *index* + 1.
 - iii. If *e*.`[[Key]]` is not empty, then
 1. If *kind* is key, let *result* be *e*.`[[Key]]`.
 2. Else if *kind* is value, let *result* be *e*.`[[Value]]`.
 3. Else,
 - a. **Assert:** *kind* is key+value.
 - b. Let *result* be ! `CreateArrayFromList`(`<< e.[[Key]], e.[[Value]] »`).
 4. Perform ? `Yield`(*result*).
 5. NOTE: the number of elements in *entries* may have changed while execution of this abstract operation was paused by `Yield`.
 6. Set *numEntries* to the number of elements of *entries*.
- e. Return **undefined**.
4. Return ! `CreateIteratorFromClosure`(*closure*, `"%MapIteratorPrototype%"`, `%MapIteratorPrototype%`).

24.1.5.2 The `%MapIteratorPrototype%` Object

The `%MapIteratorPrototype%` object:

- has properties that are inherited by all Map Iterator Objects.
- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%IteratorPrototype%`.
- has the following properties:

24.1.5.2.1 `%MapIteratorPrototype%.next()`

1. Return ? [GeneratorResume](#)(this value, empty, `"%MapIteratorPrototype%"`).

24.1.5.2.2 `%MapIteratorPrototype%` [`@@toStringTag`]

The initial value of the `@@toStringTag` property is the String value `"Map Iterator"`.

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

24.2 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.

24.2.1 The Set Constructor

The Set [constructor](#):

- is `%Set%`.
- is the initial value of the `"Set"` property of the [global object](#).
- creates and initializes a new Set object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 the Set [constructor](#) to create and initialize the subclass instance with the internal state necessary to support the **Set.prototype** built-in methods.

24.2.1.1 `Set([iterable])`

When the **Set** function is called with optional argument *iterable*, the following steps are taken:

1. If `NewTarget` is **undefined**, throw a **TypeError** exception.
2. Let *set* be ? [OrdinaryCreateFromConstructor](#)(`NewTarget`, `"%Set.prototype%"`, « `[[SetData]]` »).
3. Set *set*.`[[SetData]]` to a new empty [List](#).
4. If *iterable* is either **undefined** or **null**, return *set*.
5. Let *adder* be ? [Get](#)(*set*, `"add"`).

6. If `IsCallable(adder)` is **false**, throw a **TypeError** exception.
7. Let *iteratorRecord* be `? GetIterator(iterable)`.
8. Repeat,
 - a. Let *next* be `? IteratorStep(iteratorRecord)`.
 - b. If *next* is **false**, return *set*.
 - c. Let *nextValue* be `? IteratorValue(next)`.
 - d. Let *status* be `Call(adder, set, « nextValue »)`.
 - e. If *status* is an **abrupt completion**, return `? IteratorClose(iteratorRecord, status)`.

24.2.2 Properties of the Set Constructor

The Set **constructor**:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

24.2.2.1 Set.prototype

The initial value of **Set.prototype** is the **Set prototype object**.

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

24.2.2.2 get Set [@@species]

Set[@@species] is an **accessor property** whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the **"name"** property of this function is **"get [Symbol.species]"**.

NOTE

Methods that create derived collection objects should call `@@species` to determine the **constructor** to use to create the derived objects. Subclass **constructor** may over-ride `@@species` to change the default **constructor** assignment.

24.2.3 Properties of the Set Prototype Object

The *Set prototype object*:

- is `%Set.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an **ordinary object**.
- does not have a `[[SetData]]` internal slot.

24.2.3.1 Set.prototype.add (*value*)

The following steps are taken:

1. Let *S* be the **this** value.

2. Perform ? `RequireInternalSlot(S, [[SetData]])`.
3. Let *entries* be the `List` that is `S. [[SetData]]`.
4. For each element *e* of *entries*, do
 - a. If *e* is not `empty` and `SameValueZero(e, value)` is `true`, then
 - i. Return *S*.
5. If *value* is $-0_{\mathbb{F}}$, set *value* to $+0_{\mathbb{F}}$.
6. Append *value* as the last element of *entries*.
7. Return *S*.

24.2.3.2 `Set.prototype.clear ()`

The following steps are taken:

1. Let *S* be the `this` value.
2. Perform ? `RequireInternalSlot(S, [[SetData]])`.
3. Let *entries* be the `List` that is `S. [[SetData]]`.
4. For each element *e* of *entries*, do
 - a. Replace the element of *entries* whose value is *e* with an element whose value is `empty`.
5. Return `undefined`.

NOTE The existing `[[SetData]] List` is preserved because there may be existing Set Iterator objects that are suspended midway through iterating over that `List`.

24.2.3.3 `Set.prototype.constructor`

The initial value of `Set.prototype.constructor` is `%Set%`.

24.2.3.4 `Set.prototype.delete (value)`

The following steps are taken:

1. Let *S* be the `this` value.
2. Perform ? `RequireInternalSlot(S, [[SetData]])`.
3. Let *entries* be the `List` that is `S. [[SetData]]`.
4. For each element *e* of *entries*, do
 - 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`.
5. 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.

24.2.3.5 `Set.prototype.entries ()`

The following steps are taken:

1. Let *S* be the **this** value.
2. Return ? [CreateSetIterator](#)(*S*, 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.

24.2.3.6 Set.prototype.forEach (*callbackfn* [, *thisArg*])

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*S*, [[SetData]]).
3. If [IsCallable](#)(*callbackfn*) is **false**, throw a **TypeError** exception.
4. Let *entries* be the [List](#) that is *S*.[[SetData]].
5. For each element *e* of *entries*, do
 - a. If *e* is not **empty**, then
 - i. Perform ? [Call](#)(*callbackfn*, *thisArg*, « *e*, *e*, *S* »).
6. Return **undefined**.

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.

callbackfn is called with three arguments: the first two arguments are a value contained in the Set. The same value is 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 call back functions used by **forEach** methods for Map and Array. For Sets, each item value is considered to be both the key and the value.

forEach does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

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 **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 **forEach** call completes. New values added after the call to **forEach** begins are visited.

24.2.3.7 Set.prototype.has (*value*)

The following steps are taken:

1. Let *S* be the **this** value.

2. Perform ? [RequireInternalSlot](#)(*S*, [[SetData]]).
3. Let *entries* be the [List](#) that is *S*.[[SetData]].
4. For each element *e* of *entries*, do
 - a. If *e* is not empty and [SameValueZero](#)(*e*, *value*) is **true**, return **true**.
5. Return **false**.

24.2.3.8 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.

24.2.3.9 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. Perform ? [RequireInternalSlot](#)(*S*, [[SetData]]).
3. Let *entries* be the [List](#) that is *S*.[[SetData]].
4. Let *count* be 0.
5. For each element *e* of *entries*, do
 - a. If *e* is not empty, set *count* to *count* + 1.
6. Return $\mathbb{F}(\textit{count})$.

24.2.3.10 Set.prototype.values ()

The following steps are taken:

1. Let *S* be the **this** value.
2. Return ? [CreateSetIterator](#)(*S*, *value*).

24.2.3.11 Set.prototype [@@iterator] ()

The initial value of the @@iterator property is the same [function object](#) as the initial value of the "values" property.

24.2.3.12 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** }.

24.2.4 Properties of Set Instances

Set instances are ordinary objects that inherit properties from the Set prototype. Set instances also have a [[SetData]] internal slot.

24.2.5 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.

24.2.5.1 CreateSetIterator (*set*, *kind*)

The abstract operation CreateSetIterator takes arguments *set* and *kind*. This operation is used to create iterator objects for Set methods that return such iterators. It performs the following steps when called:

1. **Assert:** *kind* is key+value or value.
2. Perform ? [RequireInternalSlot](#)(*set*, [[SetData]]).
3. Let *closure* be a new [Abstract Closure](#) with no parameters that captures *set* and *kind* and performs the following steps when called:
 - a. Let *index* be 0.
 - b. Let *entries* be the [List](#) that is *set*.[[SetData]].
 - c. Let *numEntries* be the number of elements of *entries*.
 - d. Repeat, while *index* < *numEntries*,
 - i. Let *e* be *entries*[*index*].
 - ii. Set *index* to *index* + 1.
 - iii. If *e* is not empty, then
 1. If *kind* is key+value, then
 - a. Perform ? [Yield](#)(! [CreateArrayFromList](#)(« *e*, *e* »)).
 2. Else,
 - a. **Assert:** *kind* is value.
 - b. Perform ? [Yield](#)(*e*).
 3. NOTE: the number of elements in *entries* may have changed while execution of this abstract operation was paused by [Yield](#).
 4. Set *numEntries* to the number of elements of *entries*.
 - e. Return **undefined**.
4. Return ! [CreateIteratorFromClosure](#)(*closure*, "%SetIteratorPrototype%", %SetIteratorPrototype%).

24.2.5.2 The %SetIteratorPrototype% Object

The %SetIteratorPrototype% object:

- has properties that are inherited by all Set Iterator Objects.
- is an [ordinary object](#).
- has a [[Prototype]] internal slot whose value is %IteratorPrototype%.
- has the following properties:

24.2.5.2.1 %SetIteratorPrototype%.next ()

1. Return ? [GeneratorResume](#)(this value, empty, "%SetIteratorPrototype%").

24.2.5.2.2 %SetIteratorPrototype% [@@toStringTag]

The initial value of the @@toStringTag property is the String value "Set Iterator".

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

24.3 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 mechanism is provided for enumerating the objects it holds as keys. In certain conditions, objects which are not [live](#) are removed as WeakMap keys, as described in [9.9.3](#).

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. This characteristic can be achieved by using an inverted per-object mapping of weak map instances to keys. Alternatively each weak map may internally store its key to value mappings but this approach 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:

Barry Hayes. 1997. Ephemérons: a new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97)*, A. Michael Berman (Ed.). ACM, New York, NY, USA, 176-183, <http://doi.acm.org/10.1145/263698.263733>.

Alexandra Barros, Roberto Ierusalimsky, Eliminating Cycles in Weak Tables. *Journal of Universal Computer Science - J.UCS*, vol. 14, no. 21, pp. 3481-3497, 2008, http://www.jucs.org/jucs_14_21/eliminating_cycles_in_weak

24.3.1 The WeakMap Constructor

The WeakMap [constructor](#):

- is `%WeakMap%`.
- is the initial value of the `"WeakMap"` property of the [global object](#).
- creates and initializes a new WeakMap object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 the WeakMap [constructor](#) to create and initialize the subclass instance with the internal state necessary to support the **WeakMap.prototype** built-in methods.

24.3.1.1 WeakMap ([*iterable*])

When the **WeakMap** function is called with optional argument *iterable*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *map* be ? [OrdinaryCreateFromConstructor](#)(NewTarget, "%WeakMap.prototype%", « [[WeakMapData]] »).
3. Set *map*.[[WeakMapData]] to a new empty [List](#).
4. If *iterable* is either **undefined** or **null**, return *map*.
5. Let *adder* be ? [Get](#)(*map*, "set").
6. Return ? [AddEntriesFromIterable](#)(*map*, *iterable*, *adder*).

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.

24.3.2 Properties of the WeakMap Constructor

The WeakMap [constructor](#):

- has a [[Prototype]] internal slot whose value is [%Function.prototype%](#).
- has the following properties:

24.3.2.1 WeakMap.prototype

The initial value of **WeakMap.prototype** is the [WeakMap prototype object](#).

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

24.3.3 Properties of the WeakMap Prototype Object

The *WeakMap prototype object*:

- is [%WeakMap.prototype%](#).
- has a [[Prototype]] internal slot whose value is [%Object.prototype%](#).
- is an [ordinary object](#).
- does not have a [[WeakMapData]] internal slot.

24.3.3.1 WeakMap.prototype.constructor

The initial value of **WeakMap.prototype.constructor** is [%WeakMap%](#).

24.3.3.2 WeakMap.prototype.delete (*key*)

The following steps are taken:

1. Let M be the **this** value.
2. Perform ? `RequireInternalSlot`(M , `[[WeakMapData]]`).
3. Let *entries* be the `List` that is M .`[[WeakMapData]]`.
4. If `Type`(*key*) is not `Object`, return **false**.
5. For each `Record` { `[[Key]]`, `[[Value]]` } p of *entries*, do
 - 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**.
6. 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.

24.3.3.3 `WeakMap.prototype.get (key)`

The following steps are taken:

1. Let M be the **this** value.
2. Perform ? `RequireInternalSlot`(M , `[[WeakMapData]]`).
3. Let *entries* be the `List` that is M .`[[WeakMapData]]`.
4. If `Type`(*key*) is not `Object`, return **undefined**.
5. For each `Record` { `[[Key]]`, `[[Value]]` } p of *entries*, do
 - a. If p .`[[Key]]` is not `empty` and `SameValue`(p .`[[Key]]`, *key*) is **true**, return p .`[[Value]]`.
6. Return **undefined**.

24.3.3.4 `WeakMap.prototype.has (key)`

The following steps are taken:

1. Let M be the **this** value.
2. Perform ? `RequireInternalSlot`(M , `[[WeakMapData]]`).
3. Let *entries* be the `List` that is M .`[[WeakMapData]]`.
4. If `Type`(*key*) is not `Object`, return **false**.
5. For each `Record` { `[[Key]]`, `[[Value]]` } p of *entries*, do
 - a. If p .`[[Key]]` is not `empty` and `SameValue`(p .`[[Key]]`, *key*) is **true**, return **true**.
6. Return **false**.

24.3.3.5 `WeakMap.prototype.set (key, value)`

The following steps are taken:

1. Let M be the **this** value.
2. Perform ? `RequireInternalSlot`(M , `[[WeakMapData]]`).
3. Let *entries* be the `List` that is M .`[[WeakMapData]]`.

4. If `Type(key)` is not `Object`, throw a `TypeError` exception.
5. For each `Record` `{ [[Key]], [[Value]] }` `p` of `entries`, do
 - a. If `p.[[Key]]` is not `empty` and `SameValue(p.[[Key]], key)` is `true`, then
 - i. Set `p.[[Value]]` to `value`.
 - ii. Return `M`.
6. Let `p` be the `Record` `{ [[Key]]: key, [[Value]]: value }`.
7. Append `p` as the last element of `entries`.
8. Return `M`.

24.3.3.6 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 }`.

24.3.4 Properties of WeakMap Instances

`WeakMap` instances are ordinary objects that inherit properties from the `WeakMap` prototype. `WeakMap` instances also have a `[[WeakMapData]]` internal slot.

24.4 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 mechanism is provided for enumerating the objects it holds. In certain conditions, objects which are not `live` are removed as `WeakSet` elements, as described in 9.9.3.

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 24.3.

24.4.1 The WeakSet Constructor

The `WeakSet` constructor:

- is `%WeakSet%`.
- is the initial value of the `"WeakSet"` property of the `global` object.
- creates and initializes a new `WeakSet` object when called as a `constructor`.

- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 the WeakSet [constructor](#) to create and initialize the subclass instance with the internal state necessary to support the **WeakSet.prototype** built-in methods.

24.4.1.1 WeakSet ([*iterable*])

When the **WeakSet** function is called with optional argument *iterable*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *set* be ? [OrdinaryCreateFromConstructor](#)(NewTarget, "%WeakSet.prototype%", « [[WeakSetData]] »).
3. Set *set*.[[WeakSetData]] to a new empty [List](#).
4. If *iterable* is either **undefined** or **null**, return *set*.
5. Let *adder* be ? [Get](#)(*set*, "add").
6. If [IsCallable](#)(*adder*) is **false**, throw a **TypeError** exception.
7. Let *iteratorRecord* be ? [GetIterator](#)(*iterable*).
8. Repeat,
 - a. Let *next* be ? [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is **false**, return *set*.
 - c. Let *nextValue* be ? [IteratorValue](#)(*next*).
 - d. Let *status* be [Call](#)(*adder*, *set*, « *nextValue* »).
 - e. If *status* is an [abrupt completion](#), return ? [IteratorClose](#)(*iteratorRecord*, *status*).

24.4.2 Properties of the WeakSet Constructor

The WeakSet [constructor](#):

- has a [[Prototype]] internal slot whose value is [%Function.prototype%](#).
- has the following properties:

24.4.2.1 WeakSet.prototype

The initial value of **WeakSet.prototype** is the [WeakSet prototype object](#).

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

24.4.3 Properties of the WeakSet Prototype Object

The *WeakSet prototype object*:

- is [%WeakSet.prototype%](#).
- has a [[Prototype]] internal slot whose value is [%Object.prototype%](#).
- is an [ordinary object](#).
- does not have a [[WeakSetData]] internal slot.

24.4.3.1 WeakSet.prototype.add (*value*)

The following steps are taken:

1. Let *S* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*S*, [[WeakSetData]]).
3. If [Type](#)(*value*) is not Object, throw a **TypeError** exception.
4. Let *entries* be the [List](#) that is *S*.[[WeakSetData]].
5. For each element *e* of *entries*, do
 - a. If *e* is not empty and [SameValue](#)(*e*, *value*) is **true**, then
 - i. Return *S*.
6. Append *value* as the last element of *entries*.
7. Return *S*.

24.4.3.2 WeakSet.prototype.constructor

The initial value of **WeakSet.prototype.constructor** is the [%WeakSet%](#) intrinsic object.

24.4.3.3 WeakSet.prototype.delete (*value*)

The following steps are taken:

1. Let *S* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*S*, [[WeakSetData]]).
3. If [Type](#)(*value*) is not Object, return **false**.
4. Let *entries* be the [List](#) that is *S*.[[WeakSetData]].
5. For each element *e* of *entries*, do
 - a. If *e* is not empty and [SameValue](#)(*e*, *value*) is **true**, then
 - i. Replace the element of *entries* whose value is *e* with an element whose value is empty.
 - ii. Return **true**.
6. Return **false**.

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.

24.4.3.4 WeakSet.prototype.has (*value*)

The following steps are taken:

1. Let *S* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*S*, [[WeakSetData]]).
3. Let *entries* be the [List](#) that is *S*.[[WeakSetData]].
4. If [Type](#)(*value*) is not Object, return **false**.
5. For each element *e* of *entries*, do
 - a. If *e* is not empty and [SameValue](#)(*e*, *value*) is **true**, return **true**.
6. Return **false**.

24.4.3.5 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` }.

24.4.4 Properties of WeakSet Instances

WeakSet instances are ordinary objects that inherit properties from the WeakSet prototype. WeakSet instances also have a `[[WeakSetData]]` internal slot.

25 Structured Data

25.1 ArrayBuffer Objects

25.1.1 Notation

The descriptions below in this section, [25.4](#), and [29](#) use the read-modify-write modification function internal data structure.

A *read-modify-write modification function* is a mathematical function that is notationally represented as an abstract closure that takes two Lists of byte values as arguments and returns a [List](#) of byte values. These abstract closures satisfy all of the following properties:

- They perform all their algorithm steps atomically.
- Their individual algorithm steps are not observable.

NOTE

To aid verifying that a read-modify-write modification function's algorithm steps constitute a pure, mathematical function, the following editorial conventions are recommended:

- They do not access, directly or transitively via invoked [abstract operations](#) and abstract closures, any language or specification values except their parameters and captured values.
- They do not return completion values.

25.1.2 Abstract Operations For ArrayBuffer Objects

25.1.2.1 AllocateArrayBuffer (*constructor*, *byteLength*)

The abstract operation AllocateArrayBuffer takes arguments *constructor* and *byteLength* (a non-negative [integer](#)). It is used to create an ArrayBuffer object. It performs the following steps when called:

1. Let *obj* be ? [OrdinaryCreateFromConstructor](#)(*constructor*, "%ArrayBuffer.prototype%", « `[[ArrayBufferData]]`, `[[ArrayBufferByteLength]]`, `[[ArrayBufferDetachKey]]` »).
2. Let *block* be ? [CreateByteDataBlock](#)(*byteLength*).
3. Set *obj*.`[[ArrayBufferData]]` to *block*.
4. Set *obj*.`[[ArrayBufferByteLength]]` to *byteLength*.
5. Return *obj*.

25.1.2.2 IsDetachedBuffer (*arrayBuffer*)

The abstract operation IsDetachedBuffer takes argument *arrayBuffer*. It performs the following steps when called:

1. **Assert:** `Type(arrayBuffer)` is Object and it has an `[[ArrayBufferData]]` internal slot.
2. If `arrayBuffer.[[ArrayBufferData]]` is **null**, return **true**.
3. Return **false**.

25.1.2.3 DetachArrayBuffer (*arrayBuffer* [, *key*])

The abstract operation DetachArrayBuffer takes argument *arrayBuffer* and optional argument *key*. It performs the following steps when called:

1. **Assert:** `Type(arrayBuffer)` is Object and it has `[[ArrayBufferData]]`, `[[ArrayBufferByteLength]]`, and `[[ArrayBufferDetachKey]]` internal slots.
2. **Assert:** `IsSharedArrayBuffer(arrayBuffer)` is **false**.
3. If *key* is not present, set *key* to **undefined**.
4. If `SameValue(arrayBuffer.[[ArrayBufferDetachKey]], key)` is **false**, throw a **TypeError** exception.
5. Set `arrayBuffer.[[ArrayBufferData]]` to **null**.
6. Set `arrayBuffer.[[ArrayBufferByteLength]]` to 0.
7. 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 use the DetachArrayBuffer abstract operation. However, an ECMAScript **host** or implementation may define such operations.

25.1.2.4 CloneArrayBuffer (*srcBuffer*, *srcByteOffset*, *srcLength*, *cloneConstructor*)

The abstract operation CloneArrayBuffer takes arguments *srcBuffer* (an ArrayBuffer object), *srcByteOffset* (a non-negative **integer**), *srcLength* (a non-negative **integer**), and *cloneConstructor* (a **constructor**). It creates a new ArrayBuffer whose data is a copy of *srcBuffer*'s data over the range starting at *srcByteOffset* and continuing for *srcLength* bytes. It performs the following steps when called:

1. **Assert:** `Type(srcBuffer)` is Object and it has an `[[ArrayBufferData]]` internal slot.
2. **Assert:** `IsConstructor(cloneConstructor)` is **true**.
3. Let *targetBuffer* be ? `AllocateArrayBuffer(cloneConstructor, srcLength)`.
4. If `IsDetachedBuffer(srcBuffer)` is **true**, throw a **TypeError** exception.
5. Let *srcBlock* be `srcBuffer.[[ArrayBufferData]]`.
6. Let *targetBlock* be `targetBuffer.[[ArrayBufferData]]`.
7. Perform `CopyDataBlockBytes(targetBlock, 0, srcBlock, srcByteOffset, srcLength)`.
8. Return *targetBuffer*.

25.1.2.5 IsUnsignedElementType (*type*)

The abstract operation IsUnsignedElementType takes argument *type*. It verifies if the argument *type* is an unsigned **TypedArray element type**. It performs the following steps when called:

1. If *type* is `Uint8`, `Uint8C`, `Uint16`, `Uint32`, or `BigUint64`, return **true**.

2. Return **false**.

25.1.2.6 `IsUnclampedIntegerElementType (type)`

The abstract operation `IsUnclampedIntegerElementType` takes argument *type*. It verifies if the argument *type* is an `Integer TypedArray element type` not including `Uint8C`. It performs the following steps when called:

1. If *type* is `Int8`, `Uint8`, `Int16`, `Uint16`, `Int32`, or `Uint32`, return **true**.
2. Return **false**.

25.1.2.7 `IsBigIntElementType (type)`

The abstract operation `IsBigIntElementType` takes argument *type*. It verifies if the argument *type* is a `BigInt TypedArray element type`. It performs the following steps when called:

1. If *type* is `BigUint64` or `BigInt64`, return **true**.
2. Return **false**.

25.1.2.8 `IsNoTearConfiguration (type, order)`

The abstract operation `IsNoTearConfiguration` takes arguments *type* and *order*. It performs the following steps when called:

1. If `! IsUnclampedIntegerElementType(type)` is **true**, return **true**.
2. If `! IsBigIntElementType(type)` is **true** and *order* is not `Init` or `Unordered`, return **true**.
3. Return **false**.

25.1.2.9 `RawBytesToNumeric (type, rawBytes, isLittleEndian)`

The abstract operation `RawBytesToNumeric` takes arguments *type* (a `TypedArray element type`), *rawBytes* (a `List`), and *isLittleEndian* (a Boolean). It performs the following steps when called:

1. Let *elementSize* be the Element Size value specified in [Table 60](#) for Element Type *type*.
2. If *isLittleEndian* is **false**, reverse the order of the elements of *rawBytes*.
3. If *type* is `Float32`, then
 - a. Let *value* be the byte elements of *rawBytes* concatenated and interpreted as a little-endian bit string encoding of an [IEEE 754-2019](#) binary32 value.
 - b. If *value* is an [IEEE 754-2019](#) binary32 NaN value, return the `NaN Number value`.
 - c. Return the `Number value` that corresponds to *value*.
4. If *type* is `Float64`, then
 - a. Let *value* be the byte elements of *rawBytes* concatenated and interpreted as a little-endian bit string encoding of an [IEEE 754-2019](#) binary64 value.
 - b. If *value* is an [IEEE 754-2019](#) binary64 NaN value, return the `NaN Number value`.
 - c. Return the `Number value` that corresponds to *value*.
5. If `! IsUnsignedElementType(type)` is **true**, then
 - a. Let *intValue* be the byte elements of *rawBytes* concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.
6. Else,
 - a. Let *intValue* be the byte elements of *rawBytes* concatenated and interpreted as a bit string encoding of a

binary little-endian two's complement number of bit length $elementSize \times 8$.

7. If `!IsBigIntElementType(type)` is **true**, return the BigInt value that corresponds to *intValue*.
8. Otherwise, return the **Number** value that corresponds to *intValue*.

25.1.2.10 GetValueFromBuffer (*arrayBuffer*, *byteIndex*, *type*, *isTypedArray*, *order* [, *isLittleEndian*])

The abstract operation GetValueFromBuffer takes arguments *arrayBuffer* (an ArrayBuffer or SharedArrayBuffer), *byteIndex* (a non-negative integer), *type* (a TypedArray element type), *isTypedArray* (a Boolean), and *order* (either SeqCst or Unordered) and optional argument *isLittleEndian* (a Boolean). It performs the following steps when called:

1. **Assert**: `IsDetachedBuffer(arrayBuffer)` is **false**.
2. **Assert**: There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *type*.
3. Let *block* be *arrayBuffer*.[[ArrayBufferData]].
4. Let *elementSize* be the Element Size value specified in Table 60 for Element Type *type*.
5. If `IsSharedArrayBuffer(arrayBuffer)` is **true**, then
 - a. Let *execution* be the [[CandidateExecution]] field of the surrounding agent's Agent Record.
 - b. Let *eventList* be the [[EventList]] field of the element in *execution*.[[EventsRecords]] whose [[AgentSignifier]] is `AgentSignifier()`.
 - c. If *isTypedArray* is **true** and `IsNoTearConfiguration(type, order)` is **true**, let *noTear* be **true**; otherwise let *noTear* be **false**.
 - d. Let *rawValue* be a List of length *elementSize* whose elements are nondeterministically chosen byte values.
 - e. NOTE: In implementations, *rawValue* is the result of a non-atomic or atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the memory model to describe observable behaviour of hardware with weak consistency.
 - f. Let *readEvent* be `ReadSharedMemory` { [[Order]]: *order*, [[NoTear]]: *noTear*, [[Block]]: *block*, [[ByteIndex]]: *byteIndex*, [[ElementSize]]: *elementSize* }.
 - g. Append *readEvent* to *eventList*.
 - h. Append **Chosen Value Record** { [[Event]]: *readEvent*, [[ChosenValue]]: *rawValue* } to *execution*.[[ChosenValues]].
6. Else, let *rawValue* be a List whose elements are bytes from *block* at indices *byteIndex* (inclusive) through *byteIndex* + *elementSize* (exclusive).
7. **Assert**: The number of elements in *rawValue* is *elementSize*.
8. If *isLittleEndian* is not present, set *isLittleEndian* to the value of the [[LittleEndian]] field of the surrounding agent's Agent Record.
9. Return `RawBytesToNumeric(type, rawValue, isLittleEndian)`.

25.1.2.11 NumericToRawBytes (*type*, *value*, *isLittleEndian*)

The abstract operation NumericToRawBytes takes arguments *type* (a TypedArray element type), *value* (a BigInt or a Number), and *isLittleEndian* (a Boolean). It performs the following steps when called:

1. If *type* is Float32, then
 - a. Let *rawBytes* be a List whose elements are the 4 bytes that are the result of converting *value* to IEEE 754-2019 binary32 format using roundTiesToEven 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, *rawBytes* may be set to any implementation chosen IEEE 754-2019 binary32 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.

2. Else if *type* is Float64, then
 - a. Let *rawBytes* be a List whose elements are the 8 bytes that are the IEEE 754-2019 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, *rawBytes* may be set to any implementation chosen IEEE 754-2019 binary64 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.
3. Else,
 - a. Let *n* be the Element Size value specified in Table 60 for Element Type *type*.
 - b. Let *convOp* be the abstract operation named in the Conversion Operation column in Table 60 for Element Type *type*.
 - c. Let *intValue* be $\mathbb{R}(\text{convOp}(\text{value}))$.
 - d. If *intValue* ≥ 0 , then
 - i. Let *rawBytes* be a List whose elements are 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 whose elements are the *n*-byte binary two'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.
4. Return *rawBytes*.

25.1.2.12 SetValueInBuffer (*arrayBuffer*, *byteIndex*, *type*, *value*, *isTypedArray*, *order* [, *isLittleEndian*])

The abstract operation SetValueInBuffer takes arguments *arrayBuffer* (an ArrayBuffer or SharedArrayBuffer), *byteIndex* (a non-negative integer), *type* (a TypedArray element type), *value* (a Number or a BigInt), *isTypedArray* (a Boolean), and *order* (one of SeqCst, Unordered, or Init) and optional argument *isLittleEndian* (a Boolean). It performs the following steps when called:

1. Assert: IsDetachedBuffer(*arrayBuffer*) is **false**.
2. Assert: There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *type*.
3. Assert: Type(*value*) is BigInt if ! IsBigIntElementType(*type*) is **true**; otherwise, Type(*value*) is Number.
4. Let *block* be *arrayBuffer*.[[ArrayBufferData]].
5. Let *elementSize* be the Element Size value specified in Table 60 for Element Type *type*.
6. If *isLittleEndian* is not present, set *isLittleEndian* to the value of the [[LittleEndian]] field of the surrounding agent's Agent Record.
7. Let *rawBytes* be NumericToRawBytes(*type*, *value*, *isLittleEndian*).
8. If IsSharedArrayBuffer(*arrayBuffer*) is **true**, then
 - a. Let *execution* be the [[CandidateExecution]] field of the surrounding agent's Agent Record.
 - b. Let *eventList* be the [[EventList]] field of the element in *execution*.[[EventsRecords]] whose [[AgentSignifier]] is AgentSignifier().
 - c. If *isTypedArray* is **true** and IsNoTearConfiguration(*type*, *order*) is **true**, let *noTear* be **true**; otherwise let *noTear* be **false**.
 - d. Append WriteSharedMemory { [[Order]]: *order*, [[NoTear]]: *noTear*, [[Block]]: *block*, [[ByteIndex]]: *byteIndex*, [[ElementSize]]: *elementSize*, [[Payload]]: *rawBytes* } to *eventList*.
9. Else, store the individual bytes of *rawBytes* into *block*, starting at *block*[*byteIndex*].
10. Return NormalCompletion(undefined).

25.1.2.13 GetModifySetValueInBuffer (*arrayBuffer*, *byteIndex*, *type*, *value*, *op* [, *isLittleEndian*])

The abstract operation `GetModifySetValueInBuffer` takes arguments *arrayBuffer* (an `ArrayBuffer` object or a `SharedArrayBuffer` object), *byteIndex* (a non-negative `integer`), *type* (a `TypedArray` element type), *value* (a `Number` or a `BigInt`), and *op* (a `read-modify-write` modification function) and optional argument *isLittleEndian* (a `Boolean`). It performs the following steps when called:

1. **Assert:** `IsDetachedBuffer(arrayBuffer)` is `false`.
2. **Assert:** There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *type*.
3. **Assert:** `Type(value)` is `BigInt` if `!IsBigIntElementType(type)` is `true`; otherwise, `Type(value)` is `Number`.
4. Let *block* be *arrayBuffer*.[[`ArrayBufferData`]].
5. Let *elementSize* be the Element Size value specified in [Table 60](#) for Element Type *type*.
6. If *isLittleEndian* is not present, set *isLittleEndian* to the value of the [[`LittleEndian`]] field of the surrounding agent's Agent Record.
7. Let *rawBytes* be `NumericToRawBytes(type, value, isLittleEndian)`.
8. If `IsSharedArrayBuffer(arrayBuffer)` is `true`, then
 - a. Let *execution* be the [[`CandidateExecution`]] field of the surrounding agent's Agent Record.
 - b. Let *eventList* be the [[`EventList`]] field of the element in *execution*.[[`EventsRecords`]] whose [[`AgentSignifier`]] is `AgentSignifier()`.
 - c. Let *rawBytesRead* be a `List` of length *elementSize* whose elements are nondeterministically chosen byte values.
 - d. NOTE: In implementations, *rawBytesRead* is the result of a load-link, of a load-exclusive, or of an operand of a read-modify-write instruction on the underlying hardware. The nondeterminism is a semantic prescription of the [memory model](#) to describe observable behaviour of hardware with weak consistency.
 - e. Let *rmwEvent* be `ReadModifyWriteSharedMemory` { [[`Order`]]: `SeqCst`, [[`NoTear`]]: `true`, [[`Block`]]: *block*, [[`ByteIndex`]]: *byteIndex*, [[`ElementSize`]]: *elementSize*, [[`Payload`]]: *rawBytes*, [[`ModifyOp`]]: *op* }.
 - f. Append *rmwEvent* to *eventList*.
 - g. Append `Chosen Value Record` { [[`Event`]]: *rmwEvent*, [[`ChosenValue`]]: *rawBytesRead* } to *execution*.[[`ChosenValues`]].
9. Else,
 - a. Let *rawBytesRead* be a `List` of length *elementSize* whose elements are the sequence of *elementSize* bytes starting with *block*[*byteIndex*].
 - b. Let *rawBytesModified* be `op(rawBytesRead, rawBytes)`.
 - c. Store the individual bytes of *rawBytesModified* into *block*, starting at *block*[*byteIndex*].
10. Return `RawBytesToNumeric(type, rawBytesRead, isLittleEndian)`.

25.1.3 The `ArrayBuffer` Constructor

The `ArrayBuffer` constructor:

- is `%ArrayBuffer%`.
- is the initial value of the "`ArrayBuffer`" property of the [global object](#).
- creates and initializes a new `ArrayBuffer` object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified `ArrayBuffer` behaviour must include a **super** call to the `ArrayBuffer` [constructor](#) to create and initialize subclass instances with the internal state necessary to support the `ArrayBuffer.prototype` built-in methods.

25.1.3.1 `ArrayBuffer` (*length*)

When the **`ArrayBuffer`** function is called with argument *length*, the following steps are taken:

1. If `NewTarget` is **undefined**, throw a **`TypeError`** exception.
2. Let *byteLength* be `? ToIndex(length)`.
3. Return `? AllocateArrayBuffer(NewTarget, byteLength)`.

25.1.4 Properties of the `ArrayBuffer` Constructor

The `ArrayBuffer` **constructor**:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

25.1.4.1 `ArrayBuffer.isView` (*arg*)

The **`isView`** function takes one argument *arg*, and performs the following steps:

1. If `Type(arg)` is not `Object`, return **false**.
2. If *arg* has a `[[ViewedArrayBuffer]]` internal slot, return **true**.
3. Return **false**.

25.1.4.2 `ArrayBuffer.prototype`

The initial value of **`ArrayBuffer.prototype`** is the `ArrayBuffer` prototype object.

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

25.1.4.3 `get ArrayBuffer` [`@@species`]

`ArrayBuffer`[`@@species`] is an **accessor property** whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the **"name"** property of this function is **"get [Symbol.species]"**.

NOTE

`ArrayBuffer` prototype methods normally use their **this** value's **constructor** to create a derived object. However, a subclass **constructor** may over-ride that default behaviour by redefining its `@@species` property.

25.1.5 Properties of the `ArrayBuffer` Prototype Object

The `ArrayBuffer` *prototype object*:

- is `%ArrayBuffer.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an **ordinary object**.
- does not have an `[[ArrayBufferData]]` or `[[ArrayBufferByteLength]]` internal slot.

25.1.5.1 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. Perform ? [RequireInternalSlot](#)(*O*, [[ArrayBufferData]]).
3. If [IsSharedArrayBuffer](#)(*O*) is **true**, throw a **TypeError** exception.
4. If [IsDetachedBuffer](#)(*O*) is **true**, return `+0F`.
5. Let *length* be *O*.[[ArrayBufferByteLength]].
6. Return `F(length)`.

25.1.5.2 `ArrayBuffer.prototype.constructor`

The initial value of `ArrayBuffer.prototype.constructor` is `%ArrayBuffer%`.

25.1.5.3 `ArrayBuffer.prototype.slice (start, end)`

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*O*, [[ArrayBufferData]]).
3. If [IsSharedArrayBuffer](#)(*O*) is **true**, throw a **TypeError** exception.
4. If [IsDetachedBuffer](#)(*O*) is **true**, throw a **TypeError** exception.
5. Let *len* be *O*.[[ArrayBufferByteLength]].
6. Let *relativeStart* be ? [ToIntegerOrInfinity](#)(*start*).
7. If *relativeStart* is `-∞`, let *first* be 0.
8. Else if *relativeStart* < 0, let *first* be `max(len + relativeStart, 0)`.
9. Else, let *first* be `min(relativeStart, len)`.
10. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ? [ToIntegerOrInfinity](#)(*end*).
11. If *relativeEnd* is `-∞`, let *final* be 0.
12. Else if *relativeEnd* < 0, let *final* be `max(len + relativeEnd, 0)`.
13. Else, let *final* be `min(relativeEnd, len)`.
14. Let *newLen* be `max(final - first, 0)`.
15. Let *ctor* be ? [SpeciesConstructor](#)(*O*, `%ArrayBuffer%`).
16. Let *new* be ? [Construct](#)(*ctor*, « `F(newLen)` »).
17. Perform ? [RequireInternalSlot](#)(*new*, [[ArrayBufferData]]).
18. If [IsSharedArrayBuffer](#)(*new*) is **true**, throw a **TypeError** exception.
19. If [IsDetachedBuffer](#)(*new*) is **true**, throw a **TypeError** exception.
20. If [SameValue](#)(*new*, *O*) is **true**, throw a **TypeError** exception.
21. If *new*.[[ArrayBufferByteLength]] < *newLen*, throw a **TypeError** exception.
22. NOTE: Side-effects of the above steps may have detached *O*.
23. If [IsDetachedBuffer](#)(*O*) is **true**, throw a **TypeError** exception.
24. Let *fromBuf* be *O*.[[ArrayBufferData]].
25. Let *toBuf* be *new*.[[ArrayBufferData]].
26. Perform [CopyDataBlockBytes](#)(*toBuf*, 0, *fromBuf*, *first*, *newLen*).
27. Return *new*.

25.1.5.4 `ArrayBuffer.prototype [@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value `"ArrayBuffer"`.

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

25.1.6 Properties of ArrayBuffer Instances

ArrayBuffer instances inherit properties from the [ArrayBuffer prototype object](#). ArrayBuffer instances each have an `[[ArrayBufferData]]` internal slot, an `[[ArrayBufferByteLength]]` internal slot, and an `[[ArrayBufferDetachKey]]` 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.

ArrayBuffer instances whose `[[ArrayBufferDetachKey]]` is set to a value other than `undefined` need to have all [DetachArrayBuffer](#) calls passing that same "detach key" as an argument, otherwise a `TypeError` will result. This internal slot is only ever set by certain embedding environments, not by algorithms in this specification.

25.2 SharedArrayBuffer Objects

25.2.1 Abstract Operations for SharedArrayBuffer Objects

25.2.1.1 AllocateSharedArrayBuffer (*constructor*, *byteLength*)

The abstract operation `AllocateSharedArrayBuffer` takes arguments *constructor* and *byteLength* (a non-negative integer). It is used to create a `SharedArrayBuffer` object. It performs the following steps when called:

1. Let *obj* be ? [OrdinaryCreateFromConstructor](#)(*constructor*, "%SharedArrayBuffer.prototype%", « `[[ArrayBufferData]]`, `[[ArrayBufferByteLength]]` »).
2. Let *block* be ? [CreateSharedByteDataBlock](#)(*byteLength*).
3. Set *obj*.`[[ArrayBufferData]]` to *block*.
4. Set *obj*.`[[ArrayBufferByteLength]]` to *byteLength*.
5. Return *obj*.

25.2.1.2 IsSharedArrayBuffer (*obj*)

The abstract operation `IsSharedArrayBuffer` takes argument *obj*. It tests whether an object is an `ArrayBuffer`, a `SharedArrayBuffer`, or a subtype of either. It performs the following steps when called:

1. **Assert:** `Type(obj)` is `Object` and it has an `[[ArrayBufferData]]` internal slot.
2. Let *bufferData* be *obj*.`[[ArrayBufferData]]`.
3. If *bufferData* is `null`, return `false`.
4. If *bufferData* is a [Data Block](#), return `false`.
5. **Assert:** *bufferData* is a [Shared Data Block](#).
6. Return `true`.

25.2.2 The SharedArrayBuffer Constructor

The `SharedArrayBuffer` [constructor](#):

- is `%SharedArrayBuffer%`.
- is the initial value of the `"SharedArrayBuffer"` property of the [global object](#), if that property is present (see below).
- creates and initializes a new `SharedArrayBuffer` object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified `SharedArrayBuffer` behaviour must include a **super** call to the `SharedArrayBuffer` [constructor](#) to create and initialize subclass instances with the internal state necessary to support the **`SharedArrayBuffer.prototype`** built-in methods.

Whenever a [host](#) does not provide concurrent access to `SharedArrayBuffer` objects it may omit the `"SharedArrayBuffer"` property of the [global object](#).

NOTE Unlike an **`ArrayBuffer`**, a **`SharedArrayBuffer`** cannot become detached, and its internal `[[ArrayBufferData]]` slot is never **`null`**.

25.2.2.1 `SharedArrayBuffer` ([*length*])

When the **`SharedArrayBuffer`** function is called with optional argument *length*, the following steps are taken:

1. If `NewTarget` is **`undefined`**, throw a **`TypeError`** exception.
2. Let *byteLength* be `? ToIndex(length)`.
3. Return `? AllocateSharedArrayBuffer(NewTarget, byteLength)`.

25.2.3 Properties of the `SharedArrayBuffer` Constructor

The `SharedArrayBuffer` [constructor](#):

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

25.2.3.1 `SharedArrayBuffer.prototype`

The initial value of **`SharedArrayBuffer.prototype`** is the [SharedArrayBuffer prototype object](#).

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

25.2.3.2 `get SharedArrayBuffer [@@species]`

`SharedArrayBuffer[@@species]` is an [accessor property](#) whose set accessor function is **`undefined`**. Its get accessor function performs the following steps:

1. Return the **`this`** value.

The value of the `"name"` property of this function is `"get [Symbol.species]"`.

25.2.4 Properties of the `SharedArrayBuffer` Prototype Object

The *SharedArrayBuffer prototype object*:

- is `%SharedArrayBuffer.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an **ordinary object**.
- does not have an `[[ArrayBufferData]]` or `[[ArrayBufferByteLength]]` internal slot.

25.2.4.1 `get SharedArrayBuffer.prototype.byteLength`

`SharedArrayBuffer.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. Perform `? RequireInternalSlot(O, [[ArrayBufferData]])`.
3. If `IsSharedArrayBuffer(O)` is **false**, throw a **TypeError** exception.
4. Let *length* be `O.[[ArrayBufferByteLength]]`.
5. Return `F(length)`.

25.2.4.2 `SharedArrayBuffer.prototype.constructor`

The initial value of `SharedArrayBuffer.prototype.constructor` is `%SharedArrayBuffer%`.

25.2.4.3 `SharedArrayBuffer.prototype.slice (start, end)`

The following steps are taken:

1. Let *O* be the **this** value.
2. Perform `? RequireInternalSlot(O, [[ArrayBufferData]])`.
3. If `IsSharedArrayBuffer(O)` is **false**, throw a **TypeError** exception.
4. Let *len* be `O.[[ArrayBufferByteLength]]`.
5. Let *relativeStart* be `? ToIntegerOrInfinity(start)`.
6. If *relativeStart* is $-\infty$, let *first* be 0.
7. Else if *relativeStart* < 0 , let *first* be `max(len + relativeStart, 0)`.
8. Else, let *first* be `min(relativeStart, len)`.
9. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be `? ToIntegerOrInfinity(end)`.
10. If *relativeEnd* is $-\infty$, let *final* be 0.
11. Else if *relativeEnd* < 0 , let *final* be `max(len + relativeEnd, 0)`.
12. Else, let *final* be `min(relativeEnd, len)`.
13. Let *newLen* be `max(final - first, 0)`.
14. Let *ctor* be `? SpeciesConstructor(O, %SharedArrayBuffer%)`.
15. Let *new* be `? Construct(ctor, « F(newLen) »)`.
16. Perform `? RequireInternalSlot(new, [[ArrayBufferData]])`.
17. If `IsSharedArrayBuffer(new)` is **false**, throw a **TypeError** exception.
18. If `new.[[ArrayBufferData]]` and `O.[[ArrayBufferData]]` are the same **Shared Data Block** values, throw a **TypeError** exception.
19. If `new.[[ArrayBufferByteLength]] < newLen`, throw a **TypeError** exception.
20. Let *fromBuf* be `O.[[ArrayBufferData]]`.
21. Let *toBuf* be `new.[[ArrayBufferData]]`.
22. Perform `CopyDataBlockBytes(toBuf, 0, fromBuf, first, newLen)`.
23. Return *new*.

25.2.4.4 SharedArrayBuffer.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value "SharedArrayBuffer".

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

25.2.5 Properties of SharedArrayBuffer Instances

SharedArrayBuffer instances inherit properties from the [SharedArrayBuffer prototype object](#). SharedArrayBuffer instances each have an [[ArrayBufferData]] internal slot and an [[ArrayBufferByteLength]] internal slot.

NOTE SharedArrayBuffer instances, unlike ArrayBuffer instances, are never detached.

25.3 DataView Objects

25.3.1 Abstract Operations For DataView Objects

25.3.1.1 GetViewValue (*view*, *requestIndex*, *isLittleEndian*, *type*)

The abstract operation GetViewValue takes arguments *view*, *requestIndex*, *isLittleEndian*, and *type*. It is used by functions on DataView instances to retrieve values from the view's buffer. It performs the following steps when called:

1. Perform ? [RequireInternalSlot](#)(*view*, [[DataView]]).
2. **Assert:** *view* has a [[ViewedArrayBuffer]] internal slot.
3. Let *getIndex* be ? [ToIndex](#)(*requestIndex*).
4. Set *isLittleEndian* to ! [ToBoolean](#)(*isLittleEndian*).
5. Let *buffer* be *view*.[[ViewedArrayBuffer]].
6. If [IsDetachedBuffer](#)(*buffer*) is **true**, throw a **TypeError** exception.
7. Let *viewOffset* be *view*.[[ByteOffset]].
8. Let *viewSize* be *view*.[[ByteLength]].
9. Let *elementSize* be the Element Size value specified in [Table 60](#) for Element Type *type*.
10. If *getIndex* + *elementSize* > *viewSize*, throw a **RangeError** exception.
11. Let *bufferIndex* be *getIndex* + *viewOffset*.
12. Return [GetValueFromBuffer](#)(*buffer*, *bufferIndex*, *type*, **false**, **Unordered**, *isLittleEndian*).

25.3.1.2 SetViewValue (*view*, *requestIndex*, *isLittleEndian*, *type*, *value*)

The abstract operation SetViewValue takes arguments *view*, *requestIndex*, *isLittleEndian*, *type*, and *value*. It is used by functions on DataView instances to store values into the view's buffer. It performs the following steps when called:

1. Perform ? [RequireInternalSlot](#)(*view*, [[DataView]]).
2. **Assert:** *view* has a [[ViewedArrayBuffer]] internal slot.
3. Let *getIndex* be ? [ToIndex](#)(*requestIndex*).
4. If ! [IsBigIntElementType](#)(*type*) is **true**, let *numberValue* be ? [ToBigInt](#)(*value*).
5. Otherwise, let *numberValue* be ? [ToNumber](#)(*value*).
6. Set *isLittleEndian* to ! [ToBoolean](#)(*isLittleEndian*).
7. Let *buffer* be *view*.[[ViewedArrayBuffer]].

8. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
9. Let `viewOffset` be `view.[[ByteOffset]]`.
10. Let `viewSize` be `view.[[ByteLength]]`.
11. Let `elementSize` be the Element Size value specified in Table 60 for Element Type `type`.
12. If `getIndex + elementSize > viewSize`, throw a **RangeError** exception.
13. Let `bufferIndex` be `getIndex + viewOffset`.
14. Return `SetValueInBuffer(buffer, bufferIndex, type, numberValue, false, Unordered, isLittleEndian)`.

25.3.2 The DataView Constructor

The DataView **constructor**:

- is `%DataView%`.
- is the initial value of the **"DataView"** property of the **global object**.
- creates and initializes a new DataView object when called as a **constructor**.
- is not intended to be called as a function and will throw an exception when called in that manner.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified DataView behaviour must include a **super** call to the DataView **constructor** to create and initialize subclass instances with the internal state necessary to support the **DataView.prototype** built-in methods.

25.3.2.1 DataView (*buffer* [, *byteOffset* [, *byteLength*]])

When the **DataView** function is called with at least one argument *buffer*, the following steps are taken:

1. If `NewTarget` is **undefined**, throw a **TypeError** exception.
2. Perform `? RequireInternalSlot(buffer, [[ArrayBufferData]])`.
3. Let `offset` be `? ToIndex(byteOffset)`.
4. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
5. Let `bufferByteLength` be `buffer.[[ArrayBufferByteLength]]`.
6. If `offset > bufferByteLength`, throw a **RangeError** exception.
7. If `byteLength` is **undefined**, then
 - a. Let `viewByteLength` be `bufferByteLength - offset`.
8. Else,
 - a. Let `viewByteLength` be `? ToIndex(byteLength)`.
 - b. If `offset + viewByteLength > bufferByteLength`, throw a **RangeError** exception.
9. Let `O` be `? OrdinaryCreateFromConstructor(NewTarget, "%DataView.prototype%", « [[DataView]], [[ViewedArrayBuffer]], [[ByteLength]], [[ByteOffset]] »)`.
10. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
11. Set `O.[[ViewedArrayBuffer]]` to `buffer`.
12. Set `O.[[ByteLength]]` to `viewByteLength`.
13. Set `O.[[ByteOffset]]` to `offset`.
14. Return `O`.

25.3.3 Properties of the DataView Constructor

The DataView **constructor**:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

25.3.3.1 `DataView.prototype`

The initial value of `DataView.prototype` is the [DataView prototype object](#).

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

25.3.4 Properties of the DataView Prototype Object

The *DataView prototype object*:

- is `%DataView.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an [ordinary object](#).
- does not have a `[[DataView]]`, `[[ViewedArrayBuffer]]`, `[[ByteLength]]`, or `[[ByteOffset]]` internal slot.

25.3.4.1 `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. Perform ? [RequireInternalSlot](#)(*O*, `[[DataView]]`).
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. Return *buffer*.

25.3.4.2 `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. Perform ? [RequireInternalSlot](#)(*O*, `[[DataView]]`).
3. **Assert**: *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. If [IsDetachedBuffer](#)(*buffer*) is **true**, throw a **TypeError** exception.
6. Let *size* be *O*.`[[ByteLength]]`.
7. Return $\mathbb{F}(\textit{size})$.

25.3.4.3 `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. Perform ? [RequireInternalSlot](#)(*O*, `[[DataView]]`).

3. **Assert:** *O* has a `[[ViewedArrayBuffer]]` internal slot.
4. Let *buffer* be *O*.`[[ViewedArrayBuffer]]`.
5. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
6. Let *offset* be *O*.`[[ByteOffset]]`.
7. Return `F(offset)`.

25.3.4.4 `DataView.prototype.constructor`

The initial value of `DataView.prototype.constructor` is `%DataView%`.

25.3.4.5 `DataView.prototype.getBigInt64 (byteOffset [, littleEndian])`

When the **getBigInt64** method is called with argument *byteOffset* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. Return ? `GetViewValue(v, byteOffset, littleEndian, BigInt64)`.

25.3.4.6 `DataView.prototype.getBigUint64 (byteOffset [, littleEndian])`

When the **getBigUint64** method is called with argument *byteOffset* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. Return ? `GetViewValue(v, byteOffset, littleEndian, BigUint64)`.

25.3.4.7 `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, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Float32)`.

25.3.4.8 `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, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Float64)`.

25.3.4.9 `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 ? `GetViewValue(v, byteOffset, true, Int8)`.

25.3.4.10 `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, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Int16)`.

25.3.4.11 `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, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Int32)`.

25.3.4.12 `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 ? `GetViewValue(v, byteOffset, true, Uint8)`.

25.3.4.13 `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 **this** value.
2. If *littleEndian* is not present, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Uint16)`.

25.3.4.14 `DataView.prototype.getUint32 (byteOffset [, littleEndian])`

When the **getUint32** 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, set *littleEndian* to **false**.
3. Return ? `GetViewValue(v, byteOffset, littleEndian, Uint32)`.

25.3.4.15 `DataView.prototype.setBigInt64 (byteOffset, value [, littleEndian])`

When the **setBigInt64** 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. Return ? `SetViewValue(v, byteOffset, littleEndian, BigInt64, value)`.

25.3.4.16 `DataView.prototype.setBigUint64 (byteOffset, value [, littleEndian])`

When the **setBigUint64** 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. Return ? `SetViewValue(v, byteOffset, littleEndian, BigUint64, value)`.

25.3.4.17 `DataView.prototype.setFloat32 (byteOffset, value [, littleEndian])`

When the **setFloat32** method is called with arguments *byteOffset* and *value* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, set *littleEndian* to **false**.
3. Return ? `SetViewValue(v, byteOffset, littleEndian, Float32, value)`.

25.3.4.18 `DataView.prototype.setFloat64 (byteOffset, value [, littleEndian])`

When the **setFloat64** method is called with arguments *byteOffset* and *value* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, set *littleEndian* to **false**.
3. Return ? `SetViewValue(v, byteOffset, littleEndian, Float64, value)`.

25.3.4.19 `DataView.prototype.setInt8 (byteOffset, value)`

When the **setInt8** method is called with arguments *byteOffset* and *value*, the following steps are taken:

1. Let *v* be the **this** value.
2. Return ? `SetViewValue(v, byteOffset, true, Int8, value)`.

25.3.4.20 `DataView.prototype.setInt16 (byteOffset, value [, littleEndian])`

When the **setInt16** method is called with arguments *byteOffset* and *value* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, set *littleEndian* to **false**.
3. Return ? `SetViewValue(v, byteOffset, littleEndian, Int16, value)`.

25.3.4.21 `DataView.prototype.setInt32 (byteOffset, value [, littleEndian])`

When the **setInt32** method is called with arguments *byteOffset* and *value* and optional argument *littleEndian*, the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, set *littleEndian* to **false**.
3. Return ? [SetViewValue](#)(*v*, *byteOffset*, *littleEndian*, **Int32**, *value*).

25.3.4.22 [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 ? [SetViewValue](#)(*v*, *byteOffset*, **true**, **Uint8**, *value*).

25.3.4.23 [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, set *littleEndian* to **false**.
3. Return ? [SetViewValue](#)(*v*, *byteOffset*, *littleEndian*, **Uint16**, *value*).

25.3.4.24 [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, set *littleEndian* to **false**.
3. Return ? [SetViewValue](#)(*v*, *byteOffset*, *littleEndian*, **Uint32**, *value*).

25.3.4.25 [DataView.prototype](#) [@@toStringTag]

The initial value of the @@toStringTag property is the String value "DataView".

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

25.3.5 Properties of DataView Instances

DataView instances are ordinary objects that inherit properties from the [DataView prototype object](#). DataView instances each have [[DataView]], [[ViewedArrayBuffer]], [[ByteLength]], and [[ByteOffset]] internal slots.

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 the [DataView constructor](#).

25.4 The Atomics Object

The Atomics object:

- is `%Atomics%`.
- is the initial value of the **"Atomics"** property of the [global object](#).
- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- does not have a `[[Construct]]` internal method; it cannot be used as a [constructor](#) with the **new** operator.
- does not have a `[[Call]]` internal method; it cannot be invoked as a function.

The Atomics object provides functions that operate indivisibly (atomically) on shared memory array cells as well as functions that let agents wait for and dispatch primitive events. When used with discipline, the Atomics functions allow multi-[agent](#) programs that communicate through shared memory to execute in a well-understood order even on parallel CPUs. The rules that govern shared-memory communication are provided by the [memory model](#), defined below.

NOTE For informative guidelines for programming and implementing shared memory in ECMAScript, please see the notes at the end of the [memory model](#) section.

25.4.1 Abstract Operations for Atomics

25.4.1.1 `ValidateIntegerTypedArray (typedArray [, waitable])`

The abstract operation `ValidateIntegerTypedArray` takes argument `typedArray` and optional argument `waitable` (a Boolean). It performs the following steps when called:

1. If `waitable` is not present, set `waitable` to **false**.
2. Let `buffer` be ? `ValidateTypedArray(typedArray)`.
3. Let `typeName` be `typedArray.[[TypedArrayName]]`.
4. Let `type` be the Element Type value in [Table 60](#) for `typeName`.
5. If `waitable` is **true**, then
 - a. If `typeName` is not **"Int32Array"** or **"BigInt64Array"**, throw a **TypeError** exception.
6. Else,
 - a. If ! `IsUnclampedIntegerElementType(type)` is **false** and ! `IsBigIntElementType(type)` is **false**, throw a **TypeError** exception.
7. Return `buffer`.

25.4.1.2 `ValidateAtomicAccess (typedArray, requestIndex)`

The abstract operation `ValidateAtomicAccess` takes arguments `typedArray` and `requestIndex`. It performs the following steps when called:

1. **Assert:** `typedArray` is an Object that has a `[[ViewedArrayBuffer]]` internal slot.
2. Let `length` be `typedArray.[[ArrayLength]]`.
3. Let `accessIndex` be ? `ToIndex(requestIndex)`.
4. **Assert:** `accessIndex` ≥ 0 .
5. If `accessIndex` \geq `length`, throw a **RangeError** exception.
6. Let `arrayTypeName` be `typedArray.[[TypedArrayName]]`.
7. Let `elementSize` be the Element Size value specified in [Table 60](#) for `arrayTypeName`.
8. Let `offset` be `typedArray.[[ByteOffset]]`.
9. Return $(accessIndex \times elementSize) + offset$.

25.4.1.3 GetWaiterList (*block*, *i*)

A *WaiterList* is a semantic object that contains an ordered list of those agents that are waiting on a location (*block*, *i*) in shared memory; *block* is a [Shared Data Block](#) and *i* a byte offset into the memory of *block*. A *WaiterList* object also optionally contains a [Synchronize event](#) denoting the previous leaving of its critical section.

Initially a *WaiterList* object has an empty list and no [Synchronize event](#).

The [agent cluster](#) has a store of *WaiterList* objects; the store is indexed by (*block*, *i*). *WaiterLists* are [agent](#)-independent: a lookup in the store of *WaiterLists* by (*block*, *i*) will result in the same *WaiterList* object in any [agent](#) in the [agent cluster](#).

Each *WaiterList* has a *critical section* that controls exclusive access to that *WaiterList* during evaluation. Only a single [agent](#) may enter a *WaiterList*'s critical section at one time. Entering and leaving a *WaiterList*'s critical section is controlled by the [abstract operations](#) [EnterCriticalSection](#) and [LeaveCriticalSection](#). Operations on a *WaiterList*—adding and removing waiting agents, traversing the list of agents, suspending and notifying agents on the list, setting and retrieving the [Synchronize event](#)—may only be performed by agents that have entered the *WaiterList*'s critical section.

The abstract operation [GetWaiterList](#) takes arguments *block* (a [Shared Data Block](#)) and *i* (a non-negative [integer](#)). It performs the following steps when called:

1. [Assert](#): *block* is a [Shared Data Block](#).
2. [Assert](#): *i* and *i* + 3 are valid byte offsets within the memory of *block*.
3. [Assert](#): *i* is divisible by 4.
4. Return the [WaiterList](#) that is referenced by the pair (*block*, *i*).

25.4.1.4 EnterCriticalSection (*WL*)

The abstract operation [EnterCriticalSection](#) takes argument *WL* (a [WaiterList](#)). It performs the following steps when called:

1. [Assert](#): The calling [agent](#) is not in the [critical section](#) for any [WaiterList](#).
2. Wait until no [agent](#) is in the [critical section](#) for *WL*, then enter the [critical section](#) for *WL* (without allowing any other [agent](#) to enter).
3. If *WL* has a [Synchronize event](#), then
 - a. NOTE: A *WL* whose [critical section](#) has been entered at least once has a [Synchronize event](#) set by [LeaveCriticalSection](#).
 - b. Let *execution* be the [\[\[CandidateExecution\]\]](#) field of the [surrounding agent's Agent Record](#).
 - c. Let *eventsRecord* be the [Agent Events Record](#) in *execution*.[\[\[EventsRecords\]\]](#) whose [\[\[AgentSignifier\]\]](#) is [AgentSignifier\(\)](#).
 - d. Let *entererEventList* be *eventsRecord*.[\[\[EventList\]\]](#).
 - e. Let *enterEvent* be a new [Synchronize event](#).
 - f. Append *enterEvent* to *entererEventList*.
 - g. Let *leaveEvent* be the [Synchronize event](#) in *WL*.
 - h. Append (*leaveEvent*, *enterEvent*) to *eventsRecord*.[\[\[AgentSynchronizesWith\]\]](#).

[EnterCriticalSection](#) has *contention* when an [agent](#) attempting to enter the [critical section](#) must wait for another [agent](#) to leave it. When there is no contention, FIFO order of [EnterCriticalSection](#) calls is observable. When there is contention, an implementation may choose an arbitrary order but may not cause an [agent](#) to wait indefinitely.

25.4.1.5 LeaveCriticalSection (*WL*)

The abstract operation LeaveCriticalSection takes argument *WL* (a WaiterList). It performs the following steps when called:

1. **Assert:** The calling agent is in the critical section for *WL*.
2. Let *execution* be the [[CandidateExecution]] field of the calling surrounding's Agent Record.
3. Let *eventsRecord* be the Agent Events Record in *execution*.[[EventsRecords]] whose [[AgentSignifier]] is AgentSignifier().
4. Let *leaverEventList* be *eventsRecord*.[[EventList]].
5. Let *leaveEvent* be a new Synchronize event.
6. Append *leaveEvent* to *leaverEventList*.
7. Set the Synchronize event in *WL* to *leaveEvent*.
8. Leave the critical section for *WL*.

25.4.1.6 AddWaiter (*WL, W*)

The abstract operation AddWaiter takes arguments *WL* (a WaiterList) and *W* (an agent signifier). It performs the following steps when called:

1. **Assert:** The calling agent is in the critical section for *WL*.
2. **Assert:** *W* is not on the list of waiters in any WaiterList.
3. Add *W* to the end of the list of waiters in *WL*.

25.4.1.7 RemoveWaiter (*WL, W*)

The abstract operation RemoveWaiter takes arguments *WL* (a WaiterList) and *W* (an agent signifier). It performs the following steps when called:

1. **Assert:** The calling agent is in the critical section for *WL*.
2. **Assert:** *W* is on the list of waiters in *WL*.
3. Remove *W* from the list of waiters in *WL*.

25.4.1.8 RemoveWaiters (*WL, c*)

The abstract operation RemoveWaiters takes arguments *WL* (a WaiterList) and *c* (a non-negative integer or $+\infty$). It performs the following steps when called:

1. **Assert:** The calling agent is in the critical section for *WL*.
2. Let *L* be a new empty List.
3. Let *S* be a reference to the list of waiters in *WL*.
4. Repeat, while $c > 0$ and *S* is not an empty List,
 - a. Let *W* be the first waiter in *S*.
 - b. Add *W* to the end of *L*.
 - c. Remove *W* from *S*.
 - d. If *c* is finite, set *c* to $c - 1$.
5. Return *L*.

25.4.1.9 SuspendAgent (*WL, W, timeout*)

The abstract operation `SuspendAgent` takes arguments *WL* (a `WaiterList`), *W* (an `agent` signifier), and *timeout* (a non-negative `integer`). It performs the following steps when called:

1. **Assert:** The calling `agent` is in the `critical section` for *WL*.
2. **Assert:** *W* is equivalent to `AgentSignifier()`.
3. **Assert:** *W* is on the list of waiters in *WL*.
4. **Assert:** `AgentCanSuspend()` is **true**.
5. Perform `LeaveCriticalSection(WL)` and suspend *W* for up to *timeout* milliseconds, performing the combined operation in such a way that a notification that arrives after the `critical section` is exited but before the suspension takes effect is not lost. *W* can notify either because the timeout expired or because it was notified explicitly by another `agent` calling `NotifyWaiter(WL, W)`, and not for any other reasons at all.
6. Perform `EnterCriticalSection(WL)`.
7. If *W* was notified explicitly by another `agent` calling `NotifyWaiter(WL, W)`, return **true**.
8. Return **false**.

25.4.1.10 `NotifyWaiter (WL, W)`

The abstract operation `NotifyWaiter` takes arguments *WL* (a `WaiterList`) and *W* (an `agent` signifier). It performs the following steps when called:

1. **Assert:** The calling `agent` is in the `critical section` for *WL*.
2. Notify the `agent` *W*.

NOTE The embedding may delay notifying *W*, e.g. for resource management reasons, but *W* must eventually be notified in order to guarantee forward progress.

25.4.1.11 `AtomicReadModifyWrite (typedArray, index, value, op)`

The abstract operation `AtomicReadModifyWrite` takes arguments *typedArray*, *index*, *value*, and *op* (a `read-modify-write modification function`). *op* takes two `List` of byte values arguments and returns a `List` of byte values. This operation atomically loads a value, combines it with another value, and stores the result of the combination. It returns the loaded value. It performs the following steps when called:

1. Let *buffer* be ? `ValidateIntegerTypedArray(typedArray)`.
2. Let *indexedPosition* be ? `ValidateAtomicAccess(typedArray, index)`.
3. Let *arrayTypeName* be *typedArray*.[[`TypedArrayName`]].
4. If *typedArray*.[[`ContentType`]] is `BigInt`, let *v* be ? `ToBigInt(value)`.
5. Otherwise, let *v* be `F(? ToIntegerOrInfinity(value))`.
6. If `IsDetachedBuffer(buffer)` is **true**, throw a `TypeError` exception.
7. **NOTE:** The above check is not redundant with the check in `ValidateIntegerTypedArray` because the call to `ToBigInt` or `ToIntegerOrInfinity` on the preceding lines can have arbitrary side effects, which could cause the buffer to become detached.
8. Let *elementType* be the Element Type value in [Table 60](#) for *arrayTypeName*.
9. Return `GetModifySetValueInBuffer(buffer, indexedPosition, elementType, v, op)`.

25.4.1.12 `ByteListBitwiseOp (op, xBytes, yBytes)`

The abstract operation `ByteListBitwiseOp` takes arguments *op* (a sequence of Unicode code points), *xBytes* (a `List` of

byte values), and *yBytes* (a [List](#) of byte values). The operation atomically performs a bitwise operation on all byte values of the arguments and returns a [List](#) of byte values. It performs the following steps when called:

1. **Assert:** *op* is **&**, **^**, or **|**.
2. **Assert:** *xBytes* and *yBytes* have the same number of elements.
3. Let *result* be a new empty [List](#).
4. Let *i* be 0.
5. For each element *xByte* of *xBytes*, do
 - a. Let *yByte* be *yBytes*[*i*].
 - b. If *op* is **&**, let *resultByte* be the result of applying the bitwise AND operation to *xByte* and *yByte*.
 - c. Else if *op* is **^**, let *resultByte* be the result of applying the bitwise exclusive OR (XOR) operation to *xByte* and *yByte*.
 - d. Else, *op* is **|**. Let *resultByte* be the result of applying the bitwise inclusive OR operation to *xByte* and *yByte*.
 - e. Set *i* to *i* + 1.
 - f. Append *resultByte* to the end of *result*.
6. Return *result*.

25.4.1.13 ByteListEqual (*xBytes*, *yBytes*)

The abstract operation ByteListEqual takes arguments *xBytes* (a [List](#) of byte values) and *yBytes* (a [List](#) of byte values). It performs the following steps when called:

1. If *xBytes* and *yBytes* do not have the same number of elements, return **false**.
2. Let *i* be 0.
3. For each element *xByte* of *xBytes*, do
 - a. Let *yByte* be *yBytes*[*i*].
 - b. If *xByte* ≠ *yByte*, return **false**.
 - c. Set *i* to *i* + 1.
4. Return **true**.

25.4.2 Atomics.add (*typedArray*, *index*, *value*)

The following steps are taken:

1. Let *type* be the Element Type value in [Table 60](#) for *typedArray*.[[TypedArrayName]].
2. Let *isLittleEndian* be the value of the [[LittleEndian]] field of the [surrounding agent's Agent Record](#).
3. Let *add* be a new [read-modify-write modification function](#) with parameters (*xBytes*, *yBytes*) that captures *type* and *isLittleEndian* and performs the following steps atomically when called:
 - a. Let *x* be [RawBytesToNumeric](#)(*type*, *xBytes*, *isLittleEndian*).
 - b. Let *y* be [RawBytesToNumeric](#)(*type*, *yBytes*, *isLittleEndian*).
 - c. Let *T* be [Type](#)(*x*).
 - d. Let *sum* be *T*::add(*x*, *y*).
 - e. Let *sumBytes* be [NumericToRawBytes](#)(*type*, *sum*, *isLittleEndian*).
 - f. **Assert:** *sumBytes*, *xBytes*, and *yBytes* have the same number of elements.
 - g. Return *sumBytes*.
4. Return ? [AtomicReadModifyWrite](#)(*typedArray*, *index*, *value*, *add*).

25.4.3 **Atomics.and** (*typedArray*, *index*, *value*)

The following steps are taken:

1. Let *and* be a new **read-modify-write modification function** with parameters (*xBytes*, *yBytes*) that captures nothing and performs the following steps atomically when called:
 - a. Return **ByteListBitwiseOp**(&, *xBytes*, *yBytes*).
2. Return ? **AtomicReadModifyWrite**(*typedArray*, *index*, *value*, *and*).

25.4.4 **Atomics.compareExchange** (*typedArray*, *index*, *expectedValue*, *replacementValue*)

The following steps are taken:

1. Let *buffer* be ? **ValidateIntegerTypedArray**(*typedArray*).
2. Let *block* be *buffer*.[[ArrayBufferData]].
3. Let *indexedPosition* be ? **ValidateAtomicAccess**(*typedArray*, *index*).
4. Let *arrayTypeName* be *typedArray*.[[TypedArrayName]].
5. If *typedArray*.[[ContentType]] is **BigInt**, then
 - a. Let *expected* be ? **ToBigInt**(*expectedValue*).
 - b. Let *replacement* be ? **ToBigInt**(*replacementValue*).
6. Else,
 - a. Let *expected* be **F**(? **ToIntegerOrInfinity**(*expectedValue*)).
 - b. Let *replacement* be **F**(? **ToIntegerOrInfinity**(*replacementValue*)).
7. If **IsDetachedBuffer**(*buffer*) is **true**, throw a **TypeError** exception.
8. NOTE: The above check is not redundant with the check in **ValidateIntegerTypedArray** because the call to **ToBigInt** or **ToIntegerOrInfinity** on the preceding lines can have arbitrary side effects, which could cause the buffer to become detached.
9. Let *elementType* be the Element Type value in **Table 60** for *arrayTypeName*.
10. Let *elementSize* be the Element Size value specified in **Table 60** for Element Type *elementType*.
11. Let *isLittleEndian* be the value of the [[LittleEndian]] field of the **surrounding agent's Agent Record**.
12. Let *expectedBytes* be **NumericToRawBytes**(*elementType*, *expected*, *isLittleEndian*).
13. Let *replacementBytes* be **NumericToRawBytes**(*elementType*, *replacement*, *isLittleEndian*).
14. If **IsSharedArrayBuffer**(*buffer*) is **true**, then
 - a. Let *execution* be the [[CandidateExecution]] field of the **surrounding agent's Agent Record**.
 - b. Let *eventList* be the [[EventList]] field of the element in *execution*.[[EventsRecords]] whose [[AgentSignifier]] is **AgentSignifier**().
 - c. Let *rawBytesRead* be a **List** of length *elementSize* whose elements are nondeterministically chosen byte values.
 - d. NOTE: In implementations, *rawBytesRead* is the result of a load-link, of a load-exclusive, or of an operand of a read-modify-write instruction on the underlying hardware. The nondeterminism is a semantic prescription of the **memory model** to describe observable behaviour of hardware with weak consistency.
 - e. NOTE: The comparison of the expected value and the read value is performed outside of the **read-modify-write modification function** to avoid needlessly strong synchronization when the expected value is not equal to the read value.
 - f. If **ByteListEqual**(*rawBytesRead*, *expectedBytes*) is **true**, then
 - i. Let *second* be a new **read-modify-write modification function** with parameters (*oldBytes*, *newBytes*) that captures nothing and performs the following steps atomically when called:
 1. Return *newBytes*.

- ii. Let *event* be `ReadModifyWriteSharedMemory` { `[[Order]]: SeqCst`, `[[NoTear]]: true`, `[[Block]]: block`, `[[ByteIndex]]: indexedPosition`, `[[ElementSize]]: elementSize`, `[[Payload]]: replacementBytes`, `[[ModifyOp]]: second` }.
 - g. Else,
 - i. Let *event* be `ReadSharedMemory` { `[[Order]]: SeqCst`, `[[NoTear]]: true`, `[[Block]]: block`, `[[ByteIndex]]: indexedPosition`, `[[ElementSize]]: elementSize` }.
 - h. Append *event* to *eventList*.
 - i. Append `Chosen Value Record` { `[[Event]]: event`, `[[ChosenValue]]: rawBytesRead` } to *execution*.
`[[ChosenValues]]`.
15. Else,
- a. Let *rawBytesRead* be a `List` of length *elementSize* whose elements are the sequence of *elementSize* bytes starting with `block[indexedPosition]`.
 - b. If `ByteListEqual(rawBytesRead, expectedBytes)` is **true**, then
 - i. Store the individual bytes of *replacementBytes* into *block*, starting at `block[indexedPosition]`.
16. Return `RawBytesToNumeric(elementType, rawBytesRead, isLittleEndian)`.

25.4.5 `Atomics.exchange (typedArray, index, value)`

The following steps are taken:

1. Let *second* be a new `read-modify-write modification function` with parameters (*oldBytes*, *newBytes*) that captures nothing and performs the following steps atomically when called:
 - a. Return *newBytes*.
2. Return ? `AtomicReadModifyWrite(typedArray, index, value, second)`.

25.4.6 `Atomics.isLockFree (size)`

The following steps are taken:

1. Let *n* be ? `ToIntegerOrInfinity(size)`.
2. Let *AR* be the `Agent Record` of the `surrounding agent`.
3. If *n* = 1, return `AR.[[IsLockFree1]]`.
4. If *n* = 2, return `AR.[[IsLockFree2]]`.
5. If *n* = 4, return **true**.
6. If *n* = 8, return `AR.[[IsLockFree8]]`.
7. Return **false**.

NOTE

Atomics.isLockFree() is an optimization primitive. The intuition is that if the atomic step of an atomic primitive (**compareExchange**, **load**, **store**, **add**, **sub**, **and**, **or**, **xor**, or **exchange**) on a datum of size n bytes will be performed without the calling agent acquiring a lock outside the n bytes comprising the datum, then **Atomics.isLockFree(n)** will return **true**. High-performance algorithms will use **Atomics.isLockFree** to determine whether to use locks or atomic operations in critical sections. If an atomic primitive is not lock-free then it is often more efficient for an algorithm to provide its own locking.

Atomics.isLockFree(4) always returns **true** as that can be supported on all known relevant hardware. Being able to assume this will generally simplify programs.

Regardless of the value of **Atomics.isLockFree**, all atomic operations are guaranteed to be atomic. For example, they will never have a visible operation take place in the middle of the operation (e.g., "tearing").

25.4.7 **Atomics.load** (*typedArray*, *index*)

The following steps are taken:

1. Let *buffer* be ? **ValidateIntegerTypedArray**(*typedArray*).
2. Let *indexedPosition* be ? **ValidateAtomicAccess**(*typedArray*, *index*).
3. If **IsDetachedBuffer**(*buffer*) is **true**, throw a **TypeError** exception.
4. NOTE: The above check is not redundant with the check in **ValidateIntegerTypedArray** because the call to **ValidateAtomicAccess** on the preceding line can have arbitrary side effects, which could cause the buffer to become detached.
5. Let *arrayTypeName* be *typedArray*.[[TypedArrayName]].
6. Let *elementType* be the Element Type value in Table 60 for *arrayTypeName*.
7. Return **GetValueFromBuffer**(*buffer*, *indexedPosition*, *elementType*, **true**, SeqCst).

25.4.8 **Atomics.or** (*typedArray*, *index*, *value*)

The following steps are taken:

1. Let *or* be a new **read-modify-write modification function** with parameters (*xBytes*, *yBytes*) that captures nothing and performs the following steps atomically when called:
 - a. Return **ByteListBitwiseOp**(**I**, *xBytes*, *yBytes*).
2. Return ? **AtomicReadModifyWrite**(*typedArray*, *index*, *value*, *or*).

25.4.9 **Atomics.store** (*typedArray*, *index*, *value*)

The following steps are taken:

1. Let *buffer* be ? **ValidateIntegerTypedArray**(*typedArray*).
2. Let *indexedPosition* be ? **ValidateAtomicAccess**(*typedArray*, *index*).
3. Let *arrayTypeName* be *typedArray*.[[TypedArrayName]].
4. If *arrayTypeName* is **"BigUint64Array"** or **"BigInt64Array"**, let *v* be ? **ToBigInt**(*value*).
5. Otherwise, let *v* be **F**(? **ToIntegerOrInfinity**(*value*)).

6. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
7. NOTE: The above check is not redundant with the check in `ValidateIntegerTypedArray` because the call to `ToBigInt` or `ToIntegerOrInfinity` on the preceding lines can have arbitrary side effects, which could cause the buffer to become detached.
8. Let *elementType* be the Element Type value in Table 60 for *arrayTypeName*.
9. Perform `SetValueInBuffer(buffer, indexedPosition, elementType, v, true, SeqCst)`.
10. Return *v*.

25.4.10 `Atomics.sub` (*typedArray*, *index*, *value*)

The following steps are taken:

1. Let *type* be the Element Type value in Table 60 for *typedArray*.[[TypedArrayName]].
2. Let *isLittleEndian* be the value of the [[LittleEndian]] field of the surrounding agent's Agent Record.
3. Let *subtract* be a new read-modify-write modification function with parameters (*xBytes*, *yBytes*) that captures *type* and *isLittleEndian* and performs the following steps atomically when called:
 - a. Let *x* be `RawBytesToNumeric(type, xBytes, isLittleEndian)`.
 - b. Let *y* be `RawBytesToNumeric(type, yBytes, isLittleEndian)`.
 - c. Let *T* be `Type(x)`.
 - d. Let *difference* be `T::subtract(x, y)`.
 - e. Let *differenceBytes* be `NumericToRawBytes(type, difference, isLittleEndian)`.
 - f. **Assert**: *differenceBytes*, *xBytes*, and *yBytes* have the same number of elements.
 - g. Return *differenceBytes*.
4. Return ? `AtomicReadModifyWrite(typedArray, index, value, subtract)`.

25.4.11 `Atomics.wait` (*typedArray*, *index*, *value*, *timeout*)

Atomics.wait puts the calling agent in a wait queue and puts it to sleep until it is notified or the sleep times out.

The following steps are taken:

1. Let *buffer* be ? `ValidateIntegerTypedArray(typedArray, true)`.
2. If `IsSharedArrayBuffer(buffer)` is **false**, throw a **TypeError** exception.
3. Let *indexedPosition* be ? `ValidateAtomicAccess(typedArray, index)`.
4. Let *arrayTypeName* be *typedArray*.[[TypedArrayName]].
5. If *arrayTypeName* is **"BigInt64Array"**, let *v* be ? `ToBigInt64(value)`.
6. Otherwise, let *v* be ? `ToInt32(value)`.
7. Let *q* be ? `ToNumber(timeout)`.
8. If *q* is **NaN** or $+\infty_{\mathbb{F}}$, let *t* be $+\infty$; else if *q* is $-\infty_{\mathbb{F}}$, let *t* be 0; else let *t* be `max($\mathbb{R}(q)$, 0)`.
9. Let *B* be `AgentCanSuspend()`.
10. If *B* is **false**, throw a **TypeError** exception.
11. Let *block* be *buffer*.[[ArrayBufferData]].
12. Let *WL* be `GetWaiterList(block, indexedPosition)`.
13. Perform `EnterCriticalSection(WL)`.
14. Let *elementType* be the Element Type value in Table 60 for *arrayTypeName*.
15. Let *w* be ! `GetValueFromBuffer(buffer, indexedPosition, elementType, true, SeqCst)`.
16. If *v* \neq *w*, then
 - a. Perform `LeaveCriticalSection(WL)`.
 - b. Return the String **"not-equal"**.

17. Let W be `AgentSignifier()`.
18. Perform `AddWaiter(WL, W)`.
19. Let $notified$ be `SuspendAgent(WL, W, t)`.
20. If $notified$ is **true**, then
 - a. **Assert**: W is not on the list of waiters in WL .
21. Else,
 - a. Perform `RemoveWaiter(WL, W)`.
22. Perform `LeaveCriticalSection(WL)`.
23. If $notified$ is **true**, return the String "ok".
24. Return the String "timed-out".

25.4.12 `Atomics.notify (typedArray, index, count)`

Atomics.notify notifies some agents that are sleeping in the wait queue. The following steps are taken:

1. Let $buffer$ be ? `ValidateIntegerTypedArray(typedArray, true)`.
2. Let $indexedPosition$ be ? `ValidateAtomicAccess(typedArray, index)`.
3. If $count$ is **undefined**, let c be $+\infty$.
4. Else,
 - a. Let $intCount$ be ? `ToIntegerOrInfinity(count)`.
 - b. Let c be `max(intCount, 0)`.
5. Let $block$ be $buffer.[[ArrayBufferData]]$.
6. Let $arrayTypeName$ be $typedArray.[[TypedArrayName]]$.
7. If `IsSharedArrayBuffer(buffer)` is **false**, return $+0_F$.
8. Let WL be `GetWaiterList(block, indexedPosition)`.
9. Let n be 0.
10. Perform `EnterCriticalSection(WL)`.
11. Let S be `RemoveWaiters(WL, c)`.
12. Repeat, while S is not an empty List,
 - a. Let W be the first agent in S .
 - b. Remove W from the front of S .
 - c. Perform `NotifyWaiter(WL, W)`.
 - d. Set n to $n + 1$.
13. Perform `LeaveCriticalSection(WL)`.
14. Return $F(n)$.

25.4.13 `Atomics.xor (typedArray, index, value)`

The following steps are taken:

1. Let xor be a new **read-modify-write modification function** with parameters ($xBytes, yBytes$) that captures nothing and performs the following steps atomically when called:
 - a. Return `ByteListBitwiseOp(^, xBytes, yBytes)`.
2. Return ? `AtomicReadModifyWrite(typedArray, index, value, xor)`.

25.4.14 `Atomics [@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value "Atomics".

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

25.5 The JSON Object

The JSON object:

- is `%JSON%`.
- is the initial value of the "JSON" property of the [global object](#).
- is an [ordinary object](#).
- contains two functions, **parse** and **stringify**, that are used to parse and construct JSON texts.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- does not have a `[[Construct]]` internal method; it cannot be used as a [constructor](#) with the **new** operator.
- does not have a `[[Call]]` internal method; it cannot be invoked as a function.

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 the ECMA-404 specification without any deletions or extensions to the format.

25.5.1 JSON.parse (*text* [, *reviver*])

The **parse** function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format represents literals, arrays, and objects with a syntax similar to 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 *jsonString* be ? `Tostring(text)`.
2. Parse ! `StringToCodePoints(jsonString)` as a JSON text as specified in ECMA-404. Throw a **SyntaxError** exception if it is not a valid JSON text as defined in that specification.
3. Let *scriptString* be the [string-concatenation](#) of `"("`, *jsonString*, and `");"`.
4. Let *script* be `ParseText(!StringToCodePoints(scriptString), Script)`.
5. **Assert:** *script* is a [Parse Node](#).
6. Let *completion* be the result of evaluating *script*. The extended [PropertyDefinitionEvaluation](#) semantics defined in [B.3.1](#) must not be used during the evaluation.
7. Let *unfiltered* be *completion*.`[[Value]]`.
8. **Assert:** *unfiltered* is either a String, Number, Boolean, Null, or an Object that is defined by either an *ArrayLiteral* or an *ObjectLiteral*.
9. If `IsCallable(reviver)` is **true**, then
 - a. Let *root* be ! `OrdinaryObjectCreate(%Object.prototype%)`.
 - b. Let *rootName* be the empty String.
 - c. Perform ! `CreateDataPropertyOrThrow(root, rootName, unfiltered)`.
 - d. Return ? `InternalizeJSONProperty(root, rootName, reviver)`.

10. Else,
 - a. Return *unfiltered*.

The "length" property of the **parse** function is 2_F.

NOTE Valid JSON text is a subset of the ECMAScript *PrimaryExpression* syntax. Step 2 verifies that *jsonString* conforms to that subset, and step 8 asserts that that parsing and evaluation returns a value of an appropriate type.

However, because B.3.1 applies when evaluating ECMAScript source text and does not apply during **JSON.parse**, the same source text can produce different results when evaluated as a *PrimaryExpression* rather than as JSON. Furthermore, the Early Error for duplicate "**__proto__**" properties in object literals, which likewise does not apply during **JSON.parse**, means that not all texts accepted by **JSON.parse** are valid as a *PrimaryExpression*, despite matching the grammar.

25.5.1.1 InternalizeJSONProperty (*holder*, *name*, *reviver*)

The abstract operation InternalizeJSONProperty takes arguments *holder* (an Object), *name* (a String), and *reviver* (a function object). It performs the following steps when called:

NOTE 1 This algorithm intentionally does not throw an exception if either `[[Delete]]` or `CreateDataProperty` return **false**.

1. Let *val* be ? `Get(holder, name)`.
2. If `Type(val)` is Object, then
 - a. Let *isArray* be ? `isArray(val)`.
 - b. If *isArray* is **true**, then
 - i. Let *I* be 0.
 - ii. Let *len* be ? `LengthOfArrayLike(val)`.
 - iii. Repeat, while *I* < *len*,
 1. Let *prop* be ! `ToString(F(I))`.
 2. Let *newElement* be ? `InternalizeJSONProperty(val, prop, reviver)`.
 3. If *newElement* is **undefined**, then
 - a. Perform ? `val.[[Delete]](prop)`.
 4. Else,
 - a. Perform ? `CreateDataProperty(val, prop, newElement)`.
 5. Set *I* to *I* + 1.
 - c. Else,
 - i. Let *keys* be ? `EnumerableOwnPropertyNames(val, key)`.
 - ii. For each String *P* of *keys*, do
 1. Let *newElement* be ? `InternalizeJSONProperty(val, P, reviver)`.
 2. If *newElement* is **undefined**, then
 - a. Perform ? `val.[[Delete]](P)`.
 3. Else,
 - a. Perform ? `CreateDataProperty(val, P, newElement)`.
 3. Return ? `Call(reviver, holder, « name, val »)`.

It is not permitted for a conforming implementation of **JSON.parse** to extend the JSON grammars. If an implementation wishes to support a modified or extended JSON interchange format it must do so by defining a different parse function.

NOTE 2 In the case where there are duplicate name Strings within an object, lexically preceding values for the same key shall be overwritten.

25.5.2 JSON.stringify (*value* [, *replacer* [, *space*]])

The **stringify** function returns a String in UTF-16 encoded JSON format representing an ECMAScript value, or **undefined**. 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 an inclusion 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 a new 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. Set *ReplacerFunction* to *replacer*.
 - b. Else,
 - i. Let *isArray* be ? **IsArray**(*replacer*).
 - ii. If *isArray* is **true**, then
 1. Set *PropertyList* to a new empty List.
 2. Let *len* be ? **LengthOfArrayLike**(*replacer*).
 3. Let *k* be 0.
 4. Repeat, while *k* < *len*,
 - a. Let *prop* be ! **ToString**(**F**(*k*)).
 - b. Let *v* be ? **Get**(*replacer*, *prop*).
 - c. Let *item* be **undefined**.
 - d. If **Type**(*v*) is String, set *item* to *v*.
 - e. Else if **Type**(*v*) is Number, set *item* to ! **ToString**(*v*).
 - f. Else if **Type**(*v*) is Object, then
 - i. If *v* has a **[[StringData]]** or **[[NumberData]]** internal slot, set *item* to ? **ToString**(*v*).
 - g. If *item* is not **undefined** and *item* is not currently an element of *PropertyList*, then
 - i. Append *item* to the end of *PropertyList*.
 - h. Set *k* to *k* + 1.
 5. If **Type**(*space*) is Object, then
 - a. If *space* has a **[[NumberData]]** internal slot, then
 - i. Set *space* to ? **ToNumber**(*space*).
 - b. Else if *space* has a **[[StringData]]** internal slot, then
 - i. Set *space* to ? **ToString**(*space*).

6. If `Type(space)` is `Number`, then
 - a. Let `spaceMV` be `! ToIntegerOrInfinity(space)`.
 - b. Set `spaceMV` to `min(10, spaceMV)`.
 - c. If `spaceMV < 1`, let `gap` be the empty String; otherwise let `gap` be the String value containing `spaceMV` occurrences of the code unit `0x0020` (SPACE).
7. Else if `Type(space)` is `String`, then
 - a. If the length of `space` is 10 or less, let `gap` be `space`; otherwise let `gap` be the `substring` of `space` from 0 to 10.
8. Else,
 - a. Let `gap` be the empty String.
9. Let `wrapper` be `! OrdinaryObjectCreate(%Object.prototype%)`.
10. Perform `! CreateDataPropertyOrThrow(wrapper, the empty String, value)`.
11. Let `state` be the `Record` { `[[ReplacerFunction]]: ReplacerFunction`, `[[Stack]]: stack`, `[[Indent]]: indent`, `[[Gap]]: gap`, `[[PropertyList]]: PropertyList` }.
12. Return `? SerializeJSONProperty(state, the empty String, wrapper)`.

The "length" property of the `stringify` function is 3_F.

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 QUOTATION MARK (") code units. The code units " and \ are escaped with \ prefixes. Control characters code units are replaced with escape sequences \uHHHH, or with the shorter forms, \b (BACKSPACE), \f (FORM FEED), \n (LINE FEED), \r (CARRIAGE RETURN), \t (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 U+007B (LEFT CURLY BRACKET) followed by zero or more properties, separated with a U+002C (COMMA), closed with a U+007D (RIGHT CURLY BRACKET). A property is a quoted String representing the key or [property name](#), a U+003A (COLON), and then the stringified property value. An array is rendered as an opening U+005B (LEFT SQUARE BRACKET) followed by zero or more values, separated with a U+002C (COMMA), closed with a U+005D (RIGHT SQUARE BRACKET).

25.5.2.1 SerializeJSONProperty (*state*, *key*, *holder*)

The abstract operation `SerializeJSONProperty` takes arguments *state*, *key*, and *holder*. It performs the following steps when called:

1. Let *value* be ? `Get(holder, key)`.
2. If `Type(value)` is `Object` or `BigInt`, then
 - a. Let *toJSON* be ? `GetV(value, "toJSON")`.
 - b. If `IsCallable(toJSON)` is `true`, then
 - i. Set *value* to ? `Call(toJSON, value, « key »)`.
3. If `state.[[ReplacerFunction]]` is not `undefined`, then
 - a. Set *value* to ? `Call(state.[[ReplacerFunction]], holder, « key, value »)`.
4. If `Type(value)` is `Object`, then
 - a. If *value* has a `[[NumberData]]` internal slot, then
 - i. Set *value* to ? `ToNumber(value)`.
 - b. Else if *value* has a `[[StringData]]` internal slot, then
 - i. Set *value* to ? `ToString(value)`.
 - c. Else if *value* has a `[[BooleanData]]` internal slot, then
 - i. Set *value* to `value.[[BooleanData]]`.
 - d. Else if *value* has a `[[BigIntData]]` internal slot, then
 - i. Set *value* to `value.[[BigIntData]]`.
5. If *value* is `null`, return `"null"`.
6. If *value* is `true`, return `"true"`.
7. If *value* is `false`, return `"false"`.
8. If `Type(value)` is `String`, return `QuoteJSONString(value)`.
9. If `Type(value)` is `Number`, then
 - a. If *value* is finite, return ! `ToString(value)`.
 - b. Return `"null"`.
10. If `Type(value)` is `BigInt`, throw a `TypeError` exception.
11. If `Type(value)` is `Object` and `IsCallable(value)` is `false`, then
 - a. Let *isArray* be ? `IsArray(value)`.
 - b. If *isArray* is `true`, return ? `SerializeJSONArray(state, value)`.
 - c. Return ? `SerializeJSONObject(state, value)`.
12. Return `undefined`.

25.5.2.2 QuoteJSONString (*value*)

The abstract operation `QuoteJSONString` takes argument *value*. It wraps *value* in 0x0022 (QUOTATION MARK) code units and escapes certain other code units within it. This operation interprets *value* as a sequence of UTF-16 encoded code points, as described in 6.1.4. It performs the following steps when called:

1. Let *product* be the String value consisting solely of the code unit 0x0022 (QUOTATION MARK).
2. For each code point *C* of !StringToCodePoints(*value*), do
 - a. If *C* is listed in the “Code Point” column of Table 61, then
 - i. Set *product* to the string-concatenation of *product* and the escape sequence for *C* as specified in the “Escape Sequence” column of the corresponding row.
 - b. Else if *C* has a numeric value less than 0x0020 (SPACE), or if *C* has the same numeric value as a leading surrogate or trailing surrogate, then
 - i. Let *unit* be the code unit whose numeric value is that of *C*.
 - ii. Set *product* to the string-concatenation of *product* and UnicodeEscape(*unit*).
 - c. Else,
 - i. Set *product* to the string-concatenation of *product* and !UTF16EncodeCodePoint(*C*).
3. Set *product* to the string-concatenation of *product* and the code unit 0x0022 (QUOTATION MARK).
4. Return *product*.

Table 61: JSON Single Character Escape Sequences

Code Point	Unicode Character Name	Escape Sequence
U+0008	BACKSPACE	\b
U+0009	CHARACTER TABULATION	\t
U+000A	LINE FEED (LF)	\n
U+000C	FORM FEED (FF)	\f
U+000D	CARRIAGE RETURN (CR)	\r
U+0022	QUOTATION MARK	\"
U+005C	REVERSE SOLIDUS	\\

25.5.2.3 UnicodeEscape (*C*)

The abstract operation UnicodeEscape takes argument *C* (a code unit). It represents *C* as a Unicode escape sequence. It performs the following steps when called:

1. Let *n* be the numeric value of *C*.
2. **Assert:** $n \leq 0xFFFF$.
3. Return the string-concatenation of:
 - the code unit 0x005C (REVERSE SOLIDUS)
 - "u"
 - the String representation of *n*, formatted as a four-digit lowercase hexadecimal number, padded to the left with zeroes if necessary

25.5.2.4 SerializeJSONObject (*state*, *value*)

The abstract operation SerializeJSONObject takes arguments *state* and *value*. It serializes an object. It performs the following steps when called:

1. If *state*.[[Stack]] contains *value*, throw a **TypeError** exception because the structure is cyclical.

2. Append *value* to *state*.[[Stack]].
3. Let *stepback* be *state*.[[Indent]].
4. Set *state*.[[Indent]] to the string-concatenation of *state*.[[Indent]] and *state*.[[Gap]].
5. If *state*.[[PropertyList]] is not **undefined**, then
 - a. Let *K* be *state*.[[PropertyList]].
6. Else,
 - a. Let *K* be ? `EnumerableOwnPropertyNames`(*value*, *key*).
7. Let *partial* be a new empty List.
8. For each element *P* of *K*, do
 - a. Let *strP* be ? `SerializeJSONProperty`(*state*, *P*, *value*).
 - b. If *strP* is not **undefined**, then
 - i. Let *member* be `QuoteJSONString`(*P*).
 - ii. Set *member* to the string-concatenation of *member* and ":".
 - iii. If *state*.[[Gap]] is not the empty String, then
 1. Set *member* to the string-concatenation of *member* and the code unit 0x0020 (SPACE).
 - iv. Set *member* to the string-concatenation of *member* and *strP*.
 - v. Append *member* to *partial*.
9. If *partial* is empty, then
 - a. Let *final* be "{}".
10. Else,
 - a. If *state*.[[Gap]] is the empty String, then
 - i. Let *properties* be the String value formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the code unit 0x002C (COMMA). A comma is not inserted either before the first String or after the last String.
 - ii. Let *final* be the string-concatenation of "{", *properties*, and "}".
 - b. Else,
 - i. Let *separator* be the string-concatenation of the code unit 0x002C (COMMA), the code unit 0x000A (LINE FEED), and *state*.[[Indent]].
 - ii. Let *properties* be the String value 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 string-concatenation of "{", the code unit 0x000A (LINE FEED), *state*.[[Indent]], *properties*, the code unit 0x000A (LINE FEED), *stepback*, and "}".
11. Remove the last element of *state*.[[Stack]].
12. Set *state*.[[Indent]] to *stepback*.
13. Return *final*.

25.5.2.5 `SerializeJSONArray` (*state*, *value*)

The abstract operation `SerializeJSONArray` takes arguments *state* and *value*. It serializes an array. It performs the following steps when called:

1. If *state*.[[Stack]] contains *value*, throw a **TypeError** exception because the structure is cyclical.
2. Append *value* to *state*.[[Stack]].
3. Let *stepback* be *state*.[[Indent]].
4. Set *state*.[[Indent]] to the string-concatenation of *state*.[[Indent]] and *state*.[[Gap]].
5. Let *partial* be a new empty List.
6. Let *len* be ? `LengthOfArrayLike`(*value*).

7. Let *index* be 0.
8. Repeat, while *index* < *len*,
 - a. Let *strP* be ? `SerializeJSONProperty(state, ! ToString(F(index)), value)`.
 - b. If *strP* is **undefined**, then
 - i. Append **"null"** to *partial*.
 - c. Else,
 - i. Append *strP* to *partial*.
 - d. Set *index* to *index* + 1.
9. If *partial* is empty, then
 - a. Let *final* be **"["**.
10. Else,
 - a. If *state*.[[Gap]] is the empty String, then
 - i. Let *properties* be the String value formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the code unit 0x002C (COMMA). A comma is not inserted either before the first String or after the last String.
 - ii. Let *final* be the **string-concatenation** of **"["**, *properties*, and **"]"**.
 - b. Else,
 - i. Let *separator* be the **string-concatenation** of the code unit 0x002C (COMMA), the code unit 0x000A (LINE FEED), and *state*.[[Indent]].
 - ii. Let *properties* be the String value 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 **string-concatenation** of **"["**, the code unit 0x000A (LINE FEED), *state*.[[Indent]], *properties*, the code unit 0x000A (LINE FEED), *stepback*, and **"]"**.
11. Remove the last element of *state*.[[Stack]].
12. Set *state*.[[Indent]] to *stepback*.
13. Return *final*.

NOTE

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 opening LEFT SQUARE BRACKET, elements separated by COMMA, and a closing RIGHT SQUARE BRACKET.

25.5.3 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** }.

26 Managing Memory

26.1 WeakRef Objects

A **WeakRef** is an object that is used to refer to a target object without preserving it from garbage collection. WeakRefs

can be dereferenced to allow access to the target object, if the target object hasn't been reclaimed by garbage collection.

26.1.1 The WeakRef Constructor

The *WeakRef* constructor:

- is *%WeakRef%*.
- is the initial value of the "WeakRef" property of the [global object](#).
- creates and initializes a new WeakRef object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 **WeakRef** behaviour must include a **super** call to the **WeakRef constructor** to create and initialize the subclass instance with the internal state necessary to support the **WeakRef.prototype** built-in methods.

26.1.1.1 WeakRef (*target*)

When the **WeakRef** function is called with argument *target*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. If **Type**(*target*) is not Object, throw a **TypeError** exception.
3. Let *weakRef* be ? **OrdinaryCreateFromConstructor**(NewTarget, "**%WeakRef.prototype%**", « **[[WeakRefTarget]]** »).
4. Perform ! **AddToKeptObjects**(*target*).
5. Set *weakRef*.**[[WeakRefTarget]]** to *target*.
6. Return *weakRef*.

26.1.2 Properties of the WeakRef Constructor

The *WeakRef* constructor:

- has a **[[Prototype]]** internal slot whose value is *%Function.prototype%*.
- has the following properties:

26.1.2.1 WeakRef.prototype

The initial value of **WeakRef.prototype** is the *WeakRef prototype* object.

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

26.1.3 Properties of the WeakRef Prototype Object

The *WeakRef prototype* object:

- is *%WeakRef.prototype%*.
- has a **[[Prototype]]** internal slot whose value is *%Object.prototype%*.
- is an [ordinary object](#).
- does not have a **[[WeakRefTarget]]** internal slot.

NORMATIVE OPTIONAL

26.1.3.1 WeakRef.prototype.constructor

The initial value of **WeakRef.prototype.constructor** is *%WeakRef%*.

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

26.1.3.2 WeakRef.prototype.deref ()

The following steps are taken:

1. Let *weakRef* be the **this** value.
2. Perform ? **RequireInternalSlot**(*weakRef*, **[[WeakRefTarget]]**).
3. Return ! **WeakRefDeref**(*weakRef*).

NOTE

If the **WeakRef** returns a *target* Object that is not **undefined**, then this *target* object should not be garbage collected until the current execution of ECMAScript code has completed. The **AddToKeptObjects** operation makes sure read consistency is maintained.

```
target = { foo: function() {} };  
let weakRef = new WeakRef(target);
```

```
... later ...
```

```
if (weakRef.deref()) {  
  weakRef.deref().foo();  
}
```

In the above example, if the first deref does not evaluate to **undefined** then the second deref cannot either.

26.1.3.3 WeakRef.prototype [@@toStringTag]

The initial value of the **@@toStringTag** property is the String value **"WeakRef"**.

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

26.1.4 WeakRef Abstract Operations

26.1.4.1 WeakRefDeref (*weakRef*)

The abstract operation **WeakRefDeref** takes argument *weakRef* (a **WeakRef**). It performs the following steps when called:

1. Let *target* be *weakRef*.**[[WeakRefTarget]]**.
2. If *target* is not **empty**, then
 - a. Perform ! **AddToKeptObjects**(*target*).

- b. Return *target*.
3. Return **undefined**.

NOTE This abstract operation is defined separately from `WeakRef.prototype.deref` strictly to make it possible to succinctly define liveness.

26.1.5 Properties of WeakRef Instances

`WeakRef` instances are ordinary objects that inherit properties from the `WeakRef` prototype. `WeakRef` instances also have a `[[WeakRefTarget]]` internal slot.

26.2 FinalizationRegistry Objects

A `FinalizationRegistry` is an object that manages registration and unregistration of cleanup operations that are performed when target objects are garbage collected.

26.2.1 The FinalizationRegistry Constructor

The `FinalizationRegistry` constructor:

- is `%FinalizationRegistry%`.
- is the initial value of the "FinalizationRegistry" property of the global object.
- creates and initializes a new `FinalizationRegistry` object when called as a constructor.
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 **FinalizationRegistry** behaviour must include a **super** call to the `FinalizationRegistry` constructor to create and initialize the subclass instance with the internal state necessary to support the `FinalizationRegistry.prototype` built-in methods.

26.2.1.1 FinalizationRegistry (*cleanupCallback*)

When the **FinalizationRegistry** function is called with argument *cleanupCallback*, the following steps are taken:

1. If `NewTarget` is **undefined**, throw a **TypeError** exception.
2. If `IsCallable(cleanupCallback)` is **false**, throw a **TypeError** exception.
3. Let *finalizationRegistry* be ? `OrdinaryCreateFromConstructor`(`NewTarget`, "`%FinalizationRegistry.prototype%`", « `[[Realm]]`, `[[CleanupCallback]]`, `[[Cells]]` »).
4. Let *fn* be the active function object.
5. Set *finalizationRegistry*.`[[Realm]]` to *fn*.`[[Realm]]`.
6. Set *finalizationRegistry*.`[[CleanupCallback]]` to *cleanupCallback*.
7. Set *finalizationRegistry*.`[[Cells]]` to a new empty `List`.
8. Return *finalizationRegistry*.

26.2.2 Properties of the FinalizationRegistry Constructor

The `FinalizationRegistry` constructor:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- has the following properties:

26.2.2.1 `FinalizationRegistry.prototype`

The initial value of `FinalizationRegistry.prototype` is the `FinalizationRegistry prototype` object.

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

26.2.3 Properties of the `FinalizationRegistry` Prototype Object

The `FinalizationRegistry prototype` object:

- is `%FinalizationRegistry.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an `ordinary object`.
- does not have `[[Cells]]` and `[[CleanupCallback]]` internal slots.

26.2.3.1 `FinalizationRegistry.prototype.constructor`

The initial value of `FinalizationRegistry.prototype.constructor` is `%FinalizationRegistry%`.

26.2.3.2 `FinalizationRegistry.prototype.register (target, heldValue [, unregisterToken])`

The following steps are taken:

1. Let `finalizationRegistry` be the **this** value.
2. Perform `? RequireInternalSlot(finalizationRegistry, [[Cells]])`.
3. If `Type(target)` is not `Object`, throw a `TypeError` exception.
4. If `SameValue(target, heldValue)` is `true`, throw a `TypeError` exception.
5. If `Type(unregisterToken)` is not `Object`, then
 - a. If `unregisterToken` is not `undefined`, throw a `TypeError` exception.
 - b. Set `unregisterToken` to `empty`.
6. Let `cell` be the `Record` { `[[WeakRefTarget]]`: `target`, `[[HeldValue]]`: `heldValue`, `[[UnregisterToken]]`: `unregisterToken` }.
7. Append `cell` to `finalizationRegistry. [[Cells]]`.
8. Return `undefined`.

NOTE

Based on the algorithms and definitions in this specification, `cell. [[HeldValue]]` is `live` when `cell` is in `finalizationRegistry. [[Cells]]`; however, this does not necessarily mean that `cell. [[UnregisterToken]]` or `cell. [[Target]]` are `live`. For example, registering an object with itself as its unregister token would not keep the object alive forever.

26.2.3.3 `FinalizationRegistry.prototype.unregister (unregisterToken)`

The following steps are taken:

1. Let `finalizationRegistry` be the **this** value.

2. Perform ? `RequireInternalSlot(finalizationRegistry, [[Cells]])`.
3. If `Type(unregisterToken)` is not Object, throw a **TypeError** exception.
4. Let *removed* be **false**.
5. For each **Record** { `[[WeakRefTarget]]`, `[[HeldValue]]`, `[[UnregisterToken]]` } *cell* of `finalizationRegistry.{{Cells}}`, do
 - a. If `cell.{{UnregisterToken}}` is not empty and `SameValue(cell.{{UnregisterToken}}, unregisterToken)` is **true**, then
 - i. Remove *cell* from `finalizationRegistry.{{Cells}}`.
 - ii. Set *removed* to **true**.
6. Return *removed*.

26.2.3.4 FinalizationRegistry.prototype [@@toStringTag]

The initial value of the `@@toStringTag` property is the String value "FinalizationRegistry".

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

26.2.4 Properties of FinalizationRegistry Instances

`FinalizationRegistry` instances are ordinary objects that inherit properties from the `FinalizationRegistry` prototype. `FinalizationRegistry` instances also have `[[Cells]]` and `[[CleanupCallback]]` internal slots.

27 Control Abstraction Objects

27.1 Iteration

27.1.1 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 a distinct object. There may be many separately implemented objects that conform to any interface. An individual object may conform to multiple interfaces.

27.1.1.1 The *Iterable* Interface

The *Iterable* interface includes the property described in [Table 62](#):

Table 62: *Iterable* Interface Required Properties

Property	Value	Requirements
@@iterator	A function that returns an <i>Iterator</i> object.	The returned object must conform to the <i>Iterator</i> interface.

27.1.1.2 The *Iterator* Interface

An object that implements the *Iterator* interface must include the property in Table 63. Such objects may also implement the properties in Table 64.

Table 63: *Iterator* Interface Required Properties

Property	Value	Requirements
"next"	A function that returns an <i>IteratorResult</i> object.	The returned object must conform to the <i>IteratorResult</i> interface. If a previous call to the next method of an <i>Iterator</i> has returned an <i>IteratorResult</i> object whose "done" property is true , then all subsequent calls to the next method of that object should also return an <i>IteratorResult</i> object whose "done" property is true . However, this requirement is not enforced.

NOTE 1 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 *Iterator* objects that expect to be used in such a manner must be prepared to deal with being called with no arguments.

Table 64: *Iterator* Interface Optional Properties

Property	Value	Requirements
"return"	A function that returns an <i>IteratorResult</i> object.	The returned object must conform to the <i>IteratorResult</i> interface. Invoking this method notifies the <i>Iterator</i> object that the caller does not intend to make any more next method calls to the <i>Iterator</i> . The returned <i>IteratorResult</i> object will typically have a "done" property whose value is true , and a "value" property with the value passed as the argument of the return method. However, this requirement is not enforced.
"throw"	A function that returns an <i>IteratorResult</i> object.	The returned object must conform to the <i>IteratorResult</i> interface. Invoking this method notifies the <i>Iterator</i> object that the caller has detected an error condition. The argument may be used to identify the error condition and typically will be an exception object. A typical response is to throw the value passed as the argument. If the method does not throw , the returned <i>IteratorResult</i> object will typically have a "done" property whose value is true .

NOTE 2 Typically callers of these methods should check for their existence before invoking them. Certain ECMAScript language features including **for-of**, **yield***, and array destructuring call these methods after performing an existence check. Most ECMAScript library functions that accept *Iterable* objects as arguments also conditionally call them.

27.1.1.3 The *AsyncIterable* Interface

The *AsyncIterable* interface includes the properties described in Table 65:

Table 65: *AsyncIterable* Interface Required Properties

Property	Value	Requirements
@@asyncIterator	A function that returns an <i>AsyncIterator</i> object.	The returned object must conform to the <i>AsyncIterator</i> interface.

27.1.1.4 The *AsyncIterator* Interface

An object that implements the *AsyncIterator* interface must include the properties in Table 66. Such objects may also implement the properties in Table 67.

Table 66: *AsyncIterator* Interface Required Properties

Property	Value	Requirements
"next"	A function that returns a promise for an <i>IteratorResult</i> object.	<p>The returned promise, when fulfilled, must fulfill with an object which conforms to the <i>IteratorResult</i> interface. If a previous call to the next method of an <i>AsyncIterator</i> has returned a promise for an <i>IteratorResult</i> object whose "done" property is true, then all subsequent calls to the next method of that object should also return a promise for an <i>IteratorResult</i> object whose "done" property is true. However, this requirement is not enforced.</p> <p>Additionally, the <i>IteratorResult</i> object that serves as a fulfillment value should have a "value" property whose value is not a promise (or "thenable"). However, this requirement is also not enforced.</p>

NOTE 1

Arguments may be passed to the **next** function but their interpretation and validity is dependent upon the target *AsyncIterator*. The **for-await-of** statement and other common users of *AsyncIterators* do not pass any arguments, so *AsyncIterator* objects that expect to be used in such a manner must be prepared to deal with being called with no arguments.

Table 67: *AsyncIterator* Interface Optional Properties

Property	Value	Requirements
"return"	A function that returns a promise for an <i>IteratorResult</i> object.	<p>The returned promise, when fulfilled, must fulfill with an object which conforms to the <i>IteratorResult</i> interface. Invoking this method notifies the <i>AsyncIterator</i> object that the caller does not intend to make any more next method calls to the <i>AsyncIterator</i>. The returned promise will fulfill with an <i>IteratorResult</i> object which will typically have a "done" property whose value is true, and a "value" property with the value passed as the argument of the return method. However, this requirement is not enforced.</p> <p>Additionally, the <i>IteratorResult</i> object that serves as a fulfillment value should have a "value" property whose value is not a promise (or "thenable"). If the argument value is used in the typical manner, then if it is a rejected promise, a promise rejected with the same reason should be returned; if it is a fulfilled promise, then its fulfillment value should be used as the "value" property of the returned promise's <i>IteratorResult</i> object fulfillment value. However, these requirements are also not enforced.</p>
"throw"	A function that returns a promise for an <i>IteratorResult</i> object.	<p>The returned promise, when fulfilled, must fulfill with an object which conforms to the <i>IteratorResult</i> interface. Invoking this method notifies the <i>AsyncIterator</i> object that the caller has detected an error condition. The argument may be used to identify the error condition and typically will be an exception object. A typical response is to return a rejected promise which rejects with the value passed as the argument.</p> <p>If the returned promise is fulfilled, the <i>IteratorResult</i> fulfillment value will typically have a "done" property whose value is true. Additionally, it should have a "value" property whose value is not a promise (or "thenable"), but this requirement is not enforced.</p>

NOTE 2 Typically callers of these methods should check for their existence before invoking them. Certain ECMAScript language features including **for-await-of** and **yield*** call these methods after performing an existence check.

27.1.1.5 The *IteratorResult* Interface

The *IteratorResult* interface includes the properties listed in [Table 68](#):

Table 68: *IteratorResult* Interface Properties

Property	Value	Requirements
"done"	Either true or false .	This is the result status of an <i>iterator</i> next method call. If the end of the iterator was reached " done " is true . If the end was not reached " done " is false and a value is available. If a " done " property (either own or inherited) does not exist, it is consider to have the value false .
"value"	Any ECMAScript language value .	If done is false , this is the current iteration element value. If done is true , this is the return value of the iterator, if it supplied one. If the iterator does not have a return value, " value " is undefined . In that case, the " value " property may be absent from the conforming object if it does not inherit an explicit " value " property.

27.1.2 The *%IteratorPrototype%* Object

The *%IteratorPrototype%* object:

- has a `[[Prototype]]` internal slot whose value is *%Object.prototype%*.
- 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]()))
```

27.1.2.1 *%IteratorPrototype%* [@@iterator] ()

The following steps are taken:

1. Return the **this** value.

The value of the "**name**" property of this function is "`[Symbol.iterator]`".

27.1.3 The *%AsyncIteratorPrototype%* Object

The *%AsyncIteratorPrototype%* object:

- has a `[[Prototype]]` internal slot whose value is *%Object.prototype%*.
- is an [ordinary object](#).

NOTE

All objects defined in this specification that implement the AsyncIterator interface also inherit from `%AsyncIteratorPrototype%`. ECMAScript code may also define objects that inherit from `%AsyncIteratorPrototype%`. The `%AsyncIteratorPrototype%` object provides a place where additional methods that are applicable to all async iterator objects may be added.

27.1.3.1 `%AsyncIteratorPrototype%` [`@@asyncIterator`] ()

The following steps are taken:

1. Return the **this** value.

The value of the "name" property of this function is "[Symbol.asyncIterator]".

27.1.4 Async-from-Sync Iterator Objects

An Async-from-Sync Iterator object is an async iterator that adapts a specific synchronous iterator. There is not a named `constructor` for Async-from-Sync Iterator objects. Instead, Async-from-Sync iterator objects are created by the `CreateAsyncFromSyncIterator` abstract operation as needed.

27.1.4.1 `CreateAsyncFromSyncIterator` (*`syncIteratorRecord`*)

The abstract operation `CreateAsyncFromSyncIterator` takes argument *`syncIteratorRecord`*. It is used to create an async iterator `Record` from a synchronous iterator `Record`. It performs the following steps when called:

1. Let *`asyncIterator`* be ! `OrdinaryObjectCreate(%AsyncFromSyncIteratorPrototype%, « [[SyncIteratorRecord]] »)`.
2. Set *`asyncIterator`*.`[[SyncIteratorRecord]]` to *`syncIteratorRecord`*.
3. Let *`nextMethod`* be ! `Get(asyncIterator, "next")`.
4. Let *`iteratorRecord`* be the `Record` { `[[Iterator]]`: *`asyncIterator`*, `[[NextMethod]]`: *`nextMethod`*, `[[Done]]`: **false** }.
5. Return *`iteratorRecord`*.

27.1.4.2 The `%AsyncFromSyncIteratorPrototype%` Object

The `%AsyncFromSyncIteratorPrototype%` object:

- has properties that are inherited by all Async-from-Sync Iterator Objects.
- is an `ordinary object`.
- has a `[[Prototype]]` internal slot whose value is `%AsyncIteratorPrototype%`.
- has the following properties:

27.1.4.2.1 `%AsyncFromSyncIteratorPrototype%.next` ([*`value`*])

1. Let *`O`* be the **this** value.
2. **Assert**: `Type(O)` is Object and *`O`* has a `[[SyncIteratorRecord]]` internal slot.
3. Let *`promiseCapability`* be ! `NewPromiseCapability(%Promise%)`.
4. Let *`syncIteratorRecord`* be *`O`*.`[[SyncIteratorRecord]]`.
5. If *`value`* is present, then
 - a. Let *`result`* be `IteratorNext(syncIteratorRecord, value)`.
6. Else,

- a. Let *result* be `IteratorNext(syncIteratorRecord)`.
7. `IfAbruptRejectPromise(result, promiseCapability)`.
8. Return `! AsyncFromSyncIteratorContinuation(result, promiseCapability)`.

27.1.4.2.2 %AsyncFromSyncIteratorPrototype%.return ([*value*])

1. Let *O* be the **this** value.
2. **Assert:** `Type(O)` is Object and *O* has a `[[SyncIteratorRecord]]` internal slot.
3. Let *promiseCapability* be `! NewPromiseCapability(%Promise%)`.
4. Let *syncIterator* be `O.[[SyncIteratorRecord]].[[Iterator]]`.
5. Let *return* be `GetMethod(syncIterator, "return")`.
6. `IfAbruptRejectPromise(return, promiseCapability)`.
7. If *return* is **undefined**, then
 - a. Let *iterResult* be `! CreateIterResultObject(value, true)`.
 - b. Perform `! Call(promiseCapability.[[Resolve]], undefined, « iterResult »)`.
 - c. Return `promiseCapability.[[Promise]]`.
8. If *value* is present, then
 - a. Let *result* be `Call(return, syncIterator, « value »)`.
9. Else,
 - a. Let *result* be `Call(return, syncIterator)`.
10. `IfAbruptRejectPromise(result, promiseCapability)`.
11. If `Type(result)` is not Object, then
 - a. Perform `! Call(promiseCapability.[[Reject]], undefined, « a newly created TypeError object »)`.
 - b. Return `promiseCapability.[[Promise]]`.
12. Return `! AsyncFromSyncIteratorContinuation(result, promiseCapability)`.

27.1.4.2.3 %AsyncFromSyncIteratorPrototype%.throw ([*value*])

NOTE In this specification, *value* is always provided, but is left optional for consistency with `%AsyncFromSyncIteratorPrototype%.return ([value])`.

1. Let *O* be the **this** value.
2. **Assert:** `Type(O)` is Object and *O* has a `[[SyncIteratorRecord]]` internal slot.
3. Let *promiseCapability* be `! NewPromiseCapability(%Promise%)`.
4. Let *syncIterator* be `O.[[SyncIteratorRecord]].[[Iterator]]`.
5. Let *throw* be `GetMethod(syncIterator, "throw")`.
6. `IfAbruptRejectPromise(throw, promiseCapability)`.
7. If *throw* is **undefined**, then
 - a. Perform `! Call(promiseCapability.[[Reject]], undefined, « value »)`.
 - b. Return `promiseCapability.[[Promise]]`.
8. If *value* is present, then
 - a. Let *result* be `Call(throw, syncIterator, « value »)`.
9. Else,
 - a. Let *result* be `Call(throw, syncIterator)`.
10. `IfAbruptRejectPromise(result, promiseCapability)`.
11. If `Type(result)` is not Object, then
 - a. Perform `! Call(promiseCapability.[[Reject]], undefined, « a newly created TypeError object »)`.
 - b. Return `promiseCapability.[[Promise]]`.

12. Return ! [AsyncFromSyncIteratorContinuation](#)(*result*, *promiseCapability*).

27.1.4.2.4 Async-from-Sync Iterator Value Unwrap Functions

An async-from-sync iterator value unwrap function is an anonymous built-in function that is used by [AsyncFromSyncIteratorContinuation](#) when processing the "value" property of an *IteratorResult* object, in order to wait for its value if it is a promise and re-package the result in a new "unwrapped" *IteratorResult* object. Each async-from-sync iterator value unwrap function has a `[[Done]]` internal slot.

When an async-from-sync iterator value unwrap function is called with argument *value*, the following steps are taken:

1. Let *F* be the [active function object](#).
2. Return ! [CreateIterResultObject](#)(*value*, *F*.`[[Done]]`).

27.1.4.3 Properties of Async-from-Sync Iterator Instances

Async-from-Sync Iterator instances are ordinary objects that inherit properties from the [%AsyncFromSyncIteratorPrototype%](#) intrinsic object. Async-from-Sync Iterator instances are initially created with the internal slots listed in [Table 69](#). Async-from-Sync Iterator instances are not directly observable from ECMAScript code.

Table 69: Internal Slots of Async-from-Sync Iterator Instances

Internal Slot	Description
<code>[[SyncIteratorRecord]]</code>	A Record , of the type returned by GetIterator , representing the original synchronous iterator which is being adapted.

27.1.4.4 AsyncFromSyncIteratorContinuation (*result*, *promiseCapability*)

The abstract operation [AsyncFromSyncIteratorContinuation](#) takes arguments *result* and *promiseCapability* (a [PromiseCapability Record](#)). It performs the following steps when called:

1. Let *done* be [IteratorComplete](#)(*result*).
2. [IfAbruptRejectPromise](#)(*done*, *promiseCapability*).
3. Let *value* be [IteratorValue](#)(*result*).
4. [IfAbruptRejectPromise](#)(*value*, *promiseCapability*).
5. Let *valueWrapper* be [PromiseResolve](#)([%Promise%](#), *value*).
6. [IfAbruptRejectPromise](#)(*valueWrapper*, *promiseCapability*).
7. Let *steps* be the algorithm steps defined in [Async-from-Sync Iterator Value Unwrap Functions](#).
8. Let *length* be the number of non-optional parameters of the function definition in [Async-from-Sync Iterator Value Unwrap Functions](#).
9. Let *onFulfilled* be ! [CreateBuiltinFunction](#)(*steps*, *length*, "", « `[[Done]]` »).
10. Set *onFulfilled*.`[[Done]]` to *done*.
11. Perform ! [PerformPromiseThen](#)(*valueWrapper*, *onFulfilled*, `undefined`, *promiseCapability*).
12. Return *promiseCapability*.`[[Promise]]`.

27.2 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(f, r)** will immediately enqueue a **Job** to call the function **f**.
- A promise **p** is rejected if **p.then(f, r)** will immediately enqueue a **Job** to call the function **r**.
- 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.

27.2.1 Promise Abstract Operations

27.2.1.1 PromiseCapability Records

A *PromiseCapability Record* 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 70**.

Table 70: **PromiseCapability Record Fields**

Field Name	Value	Meaning
[[Promise]]	An object	An object that is usable as a promise.
[[Resolve]]	A function object	The function that is used to resolve the given promise object.
[[Reject]]	A function object	The function that is used to reject the given promise object.

27.2.1.1.1 IfAbruptRejectPromise (*value*, *capability*)

IfAbruptRejectPromise is a shorthand 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**, then
 - a. Perform ? **Call**(*capability*.[[Reject]], **undefined**, « *value*.[[Value]] »).
 - b. Return *capability*.[[Promise]].
2. Else if *value* is a **Completion Record**, set *value* to *value*.[[Value]].

27.2.1.2 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 **PerformPromiseThen** abstract

operation, and are used by the [Abstract Closure](#) returned by [NewPromiseReactionJob](#).

PromiseReaction records have the fields listed in [Table 71](#).

Table 71: PromiseReaction Record Fields

Field Name	Value	Meaning
[[Capability]]	A PromiseCapability Record , or undefined	The capabilities of the promise for which this record provides a reaction handler.
[[Type]]	Fulfill Reject	The [[Type]] is used when [[Handler]] is empty to allow for behaviour specific to the settlement type.
[[Handler]]	A JobCallback Record empty .	The function that should be applied to the incoming value, and whose return value will govern what happens to the derived promise. If [[Handler]] is empty , a function that depends on the value of [[Type]] will be used instead.

27.2.1.3 CreateResolvingFunctions (*promise*)

The abstract operation CreateResolvingFunctions takes argument *promise*. It performs the following steps when called:

1. Let *alreadyResolved* be the [Record](#) { [[Value]]: **false** }.
2. Let *stepsResolve* be the algorithm steps defined in [Promise Resolve Functions](#).
3. Let *lengthResolve* be the number of non-optional parameters of the function definition in [Promise Resolve Functions](#).
4. Let *resolve* be ! [CreateBuiltinFunction](#)(*stepsResolve*, *lengthResolve*, "", « [[Promise]], [[AlreadyResolved]] »).
5. Set *resolve*.[[Promise]] to *promise*.
6. Set *resolve*.[[AlreadyResolved]] to *alreadyResolved*.
7. Let *stepsReject* be the algorithm steps defined in [Promise Reject Functions](#).
8. Let *lengthReject* be the number of non-optional parameters of the function definition in [Promise Reject Functions](#).
9. Let *reject* be ! [CreateBuiltinFunction](#)(*stepsReject*, *lengthReject*, "", « [[Promise]], [[AlreadyResolved]] »).
10. Set *reject*.[[Promise]] to *promise*.
11. Set *reject*.[[AlreadyResolved]] to *alreadyResolved*.
12. Return the [Record](#) { [[Resolve]]: *resolve*, [[Reject]]: *reject* }.

27.2.1.3.1 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 is called with argument *reason*, the following steps are taken:

1. Let *F* be the [active function object](#).
2. **Assert:** *F* has a [[Promise]] internal slot whose value is an Object.
3. Let *promise* be *F*.[[Promise]].
4. Let *alreadyResolved* be *F*.[[AlreadyResolved]].
5. If *alreadyResolved*.[[Value]] is **true**, return **undefined**.

6. Set *alreadyResolved*.[[Value]] to **true**.
7. Return `RejectPromise`(*promise*, *reason*).

The "length" property of a promise reject function is 1_F.

27.2.1.3.2 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 is called with argument *resolution*, the following steps are taken:

1. Let *F* be the active function object.
2. **Assert**: *F* has a [[Promise]] internal slot whose value is an Object.
3. Let *promise* be *F*.[[Promise]].
4. Let *alreadyResolved* be *F*.[[AlreadyResolved]].
5. If *alreadyResolved*.[[Value]] is **true**, return **undefined**.
6. Set *alreadyResolved*.[[Value]] to **true**.
7. If `SameValue`(*resolution*, *promise*) is **true**, then
 - a. Let *selfResolutionError* be a newly created **TypeError** object.
 - b. Return `RejectPromise`(*promise*, *selfResolutionError*).
8. If `Type`(*resolution*) is not Object, then
 - a. Return `FulfillPromise`(*promise*, *resolution*).
9. Let *then* be `Get`(*resolution*, "then").
10. If *then* is an abrupt completion, then
 - a. Return `RejectPromise`(*promise*, *then*.[[Value]]).
11. Let *thenAction* be *then*.[[Value]].
12. If `IsCallable`(*thenAction*) is **false**, then
 - a. Return `FulfillPromise`(*promise*, *resolution*).
13. Let *thenJobCallback* be `HostMakeJobCallback`(*thenAction*).
14. Let *job* be `NewPromiseResolveThenableJob`(*promise*, *resolution*, *thenJobCallback*).
15. Perform `HostEnqueuePromiseJob`(*job*.[[Job]], *job*.[[Realm]]).
16. Return **undefined**.

The "length" property of a promise resolve function is 1_F.

27.2.1.4 FulfillPromise (*promise*, *value*)

The abstract operation FulfillPromise takes arguments *promise* and *value*. It performs the following steps when called:

1. **Assert**: The value of *promise*.[[PromiseState]] is pending.
2. Let *reactions* be *promise*.[[PromiseFulfillReactions]].
3. Set *promise*.[[PromiseResult]] to *value*.
4. Set *promise*.[[PromiseFulfillReactions]] to **undefined**.
5. Set *promise*.[[PromiseRejectReactions]] to **undefined**.
6. Set *promise*.[[PromiseState]] to fulfilled.
7. Return `TriggerPromiseReactions`(*reactions*, *value*).

27.2.1.5 NewPromiseCapability (*C*)

The abstract operation `NewPromiseCapability` takes argument `C`. It attempts to use `C` as a [constructor](#) in the fashion of the built-in `Promise` [constructor](#) to create a `Promise` object and extract its **`resolve`** and **`reject`** functions. The `Promise` object plus the **`resolve`** and **`reject`** functions are used to initialize a new [PromiseCapability Record](#). It performs the following steps when called:

1. If `IsConstructor(C)` is **false**, throw a **`TypeError`** exception.
2. NOTE: `C` is assumed to be a [constructor](#) function that supports the parameter conventions of the `Promise` [constructor](#) (see 27.2.3.1).
3. Let `promiseCapability` be the [PromiseCapability Record](#) { `[[Promise]]`: **undefined**, `[[Resolve]]`: **undefined**, `[[Reject]]`: **undefined** }.
4. Let `steps` be the algorithm steps defined in [GetCapabilitiesExecutor Functions](#).
5. Let `length` be the number of non-optional parameters of the function definition in [GetCapabilitiesExecutor Functions](#).
6. Let `executor` be `! CreateBuiltinFunction(steps, length, "", « [[Capability]] »)`.
7. Set `executor.{{Capability}}` to `promiseCapability`.
8. Let `promise` be `? Construct(C, « executor »)`.
9. If `IsCallable(promiseCapability.{{Resolve}}`) is **false**, throw a **`TypeError`** exception.
10. If `IsCallable(promiseCapability.{{Reject}}`) is **false**, throw a **`TypeError`** exception.
11. Set `promiseCapability.{{Promise}}` to `promise`.
12. Return `promiseCapability`.

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.

27.2.1.5.1 GetCapabilitiesExecutor Functions

A `GetCapabilitiesExecutor` function is an anonymous built-in function that has a `[[Capability]]` internal slot.

When a `GetCapabilitiesExecutor` function is called with arguments `resolve` and `reject`, the following steps are taken:

1. Let `F` be the [active function object](#).
2. **Assert:** `F` has a `[[Capability]]` internal slot whose value is a [PromiseCapability Record](#).
3. Let `promiseCapability` be `F.{{Capability}}`.
4. If `promiseCapability.{{Resolve}}` is not **undefined**, throw a **`TypeError`** exception.
5. If `promiseCapability.{{Reject}}` is not **undefined**, throw a **`TypeError`** exception.
6. Set `promiseCapability.{{Resolve}}` to `resolve`.
7. Set `promiseCapability.{{Reject}}` to `reject`.
8. Return **undefined**.

The **"length"** property of a `GetCapabilitiesExecutor` function is `2`.

27.2.1.6 IsPromise (x)

The abstract operation `IsPromise` takes argument `x`. It checks for the promise brand on an object. It performs the following steps when called:

1. If `Type(x)` is not `Object`, return **false**.
2. If `x` does not have a `[[PromiseState]]` internal slot, return **false**.

3. Return **true**.

27.2.1.7 **RejectPromise** (*promise*, *reason*)

The abstract operation **RejectPromise** takes arguments *promise* and *reason*. It performs the following steps when called:

1. **Assert**: The value of *promise*.[[PromiseState]] is **pending**.
2. Let *reactions* be *promise*.[[PromiseRejectReactions]].
3. Set *promise*.[[PromiseResult]] to *reason*.
4. Set *promise*.[[PromiseFulfillReactions]] to **undefined**.
5. Set *promise*.[[PromiseRejectReactions]] to **undefined**.
6. Set *promise*.[[PromiseState]] to **rejected**.
7. If *promise*.[[PromiseIsHandled]] is **false**, perform **HostPromiseRejectionTracker**(*promise*, **"reject"**).
8. Return **TriggerPromiseReactions**(*reactions*, *reason*).

27.2.1.8 **TriggerPromiseReactions** (*reactions*, *argument*)

The abstract operation **TriggerPromiseReactions** takes arguments *reactions* (a **List** of **PromiseReaction** Records) and *argument*. It enqueues a new **Job** for each record in *reactions*. Each such **Job** processes the [[Type]] and [[Handler]] of the **PromiseReaction** Record, and if the [[Handler]] is not **empty**, calls it passing the given argument. If the [[Handler]] is **empty**, the behaviour is determined by the [[Type]]. It performs the following steps when called:

1. For each element *reaction* of *reactions*, do
 - a. Let *job* be **NewPromiseReactionJob**(*reaction*, *argument*).
 - b. Perform **HostEnqueuePromiseJob**(*job*.[[Job]], *job*.[[Realm]]).
2. Return **undefined**.

27.2.1.9 **HostPromiseRejectionTracker** (*promise*, *operation*)

The **host-defined** abstract operation **HostPromiseRejectionTracker** takes arguments *promise* (a **Promise**) and *operation* (**"reject"** or **"handle"**). It allows **host** environments to track promise rejections.

An implementation of **HostPromiseRejectionTracker** must complete normally in all cases. The default implementation of **HostPromiseRejectionTracker** is to unconditionally return an empty normal completion.

NOTE 1

HostPromiseRejectionTracker is called in two scenarios:

- When a promise is rejected without any handlers, it is called with its *operation* argument set to **"reject"**.
- When a handler is added to a rejected promise for the first time, it is called with its *operation* argument set to **"handle"**.

A typical implementation of **HostPromiseRejectionTracker** might try to notify developers of unhandled rejections, while also being careful to notify them if such previous notifications are later invalidated by new handlers being attached.

If *operation* is "handle", an implementation should not hold a reference to *promise* in a way that would interfere with garbage collection. An implementation may hold a reference to *promise* if *operation* is "reject", since it is expected that rejections will be rare and not on hot code paths.

27.2.2 Promise Jobs

27.2.2.1 NewPromiseReactionJob (*reaction*, *argument*)

The abstract operation NewPromiseReactionJob takes arguments *reaction* and *argument*. It returns a new [Job Abstract Closure](#) that 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. It performs the following steps when called:

1. Let *job* be a new [Job Abstract Closure](#) with no parameters that captures *reaction* and *argument* and performs the following steps when called:
 - a. **Assert:** *reaction* is a PromiseReaction Record.
 - b. Let *promiseCapability* be *reaction*.[[Capability]].
 - c. Let *type* be *reaction*.[[Type]].
 - d. Let *handler* be *reaction*.[[Handler]].
 - e. If *handler* is empty, then
 - i. If *type* is Fulfill, let *handlerResult* be [NormalCompletion](#)(*argument*).
 - ii. Else,
 1. **Assert:** *type* is Reject.
 2. Let *handlerResult* be [ThrowCompletion](#)(*argument*).
 - f. Else, let *handlerResult* be [HostCallJobCallback](#)(*handler*, **undefined**, « *argument* »).
 - g. If *promiseCapability* is **undefined**, then
 - i. **Assert:** *handlerResult* is not an abrupt completion.
 - ii. Return [NormalCompletion](#)(empty).
 - h. **Assert:** *promiseCapability* is a PromiseCapability Record.
 - i. If *handlerResult* is an abrupt completion, then
 - i. Let *status* be [Call](#)(*promiseCapability*.[[Reject]], **undefined**, « *handlerResult*.[[Value]] »).
 - j. Else,
 - i. Let *status* be [Call](#)(*promiseCapability*.[[Resolve]], **undefined**, « *handlerResult*.[[Value]] »).
 - k. Return [Completion](#)(*status*).
2. Let *handlerRealm* be **null**.
3. If *reaction*.[[Handler]] is not empty, then
 - a. Let *getHandlerRealmResult* be [GetFunctionRealm](#)(*reaction*.[[Handler]].[[Callback]]).
 - b. If *getHandlerRealmResult* is a normal completion, set *handlerRealm* to *getHandlerRealmResult*.[[Value]].
 - c. Else, set *handlerRealm* to [the current Realm Record](#).
 - d. NOTE: *handlerRealm* is never **null** unless the handler is **undefined**. When the handler is a revoked Proxy and no ECMAScript code runs, *handlerRealm* is used to create error objects.
4. Return the [Record](#) { [[Job]]: *job*, [[Realm]]: *handlerRealm* }.

27.2.2.2 NewPromiseResolveThenableJob (*promiseToResolve*, *thenable*, *then*)

The abstract operation NewPromiseResolveThenableJob takes arguments *promiseToResolve*, *thenable*, and *then*. It performs the following steps when called:

1. Let *job* be a new [Job Abstract Closure](#) with no parameters that captures *promiseToResolve*, *thenable*, and *then* and performs the following steps when called:
 - a. Let *resolvingFunctions* be [CreateResolvingFunctions](#)(*promiseToResolve*).
 - b. Let *thenCallResult* be [HostCallJobCallback](#)(*then*, *thenable*, « *resolvingFunctions*.[[Resolve]], *resolvingFunctions*.[[Reject]] »).
 - c. If *thenCallResult* is an [abrupt completion](#), then
 - i. Let *status* be [Call](#)(*resolvingFunctions*.[[Reject]], **undefined**, « *thenCallResult*.[[Value]] »).
 - ii. Return [Completion](#)(*status*).
 - d. Return [Completion](#)(*thenCallResult*).
2. Let *getThenRealmResult* be [GetFunctionRealm](#)(*then*.[[Callback]]).
3. If *getThenRealmResult* is a normal completion, let *thenRealm* be *getThenRealmResult*.[[Value]].
4. Else, let *thenRealm* be [the current Realm Record](#).
5. NOTE: *thenRealm* is never **null**. When *then*.[[Callback]] is a revoked Proxy and no code runs, *thenRealm* is used to create error objects.
6. Return the [Record](#) { [[Job]]: *job*, [[Realm]]: *thenRealm* }.

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.

27.2.3 The Promise Constructor

The Promise [constructor](#):

- is *%Promise%*.
- is the initial value of the "Promise" property of the [global object](#).
- creates and initializes a new Promise object when called as a [constructor](#).
- is not intended to be called as a function and will throw an exception when called in that manner.
- 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 the Promise [constructor](#) to create and initialize the subclass instance with the internal state necessary to support the **Promise** and **Promise.prototype** built-in methods.

27.2.3.1 Promise (*executor*)

When the **Promise** function is called with argument *executor*, the following steps are taken:

1. If *NewTarget* is **undefined**, throw a **TypeError** exception.
2. If [IsCallable](#)(*executor*) is **false**, throw a **TypeError** exception.
3. Let *promise* be ? [OrdinaryCreateFromConstructor](#)(*NewTarget*, "%**Promise.prototype**", « [[PromiseState]], [[PromiseResult]], [[PromiseFulfillReactions]], [[PromiseRejectReactions]], [[PromiseIsHandled]] »).
4. Set *promise*.[[PromiseState]] to **pending**.
5. Set *promise*.[[PromiseFulfillReactions]] to a new empty [List](#).
6. Set *promise*.[[PromiseRejectReactions]] to a new empty [List](#).
7. Set *promise*.[[PromiseIsHandled]] to **false**.
8. Let *resolvingFunctions* be [CreateResolvingFunctions](#)(*promise*).
9. Let *completion* be [Call](#)(*executor*, **undefined**, « *resolvingFunctions*.[[Resolve]], *resolvingFunctions*.[[Reject]] »).

10. If *completion* is an **abrupt completion**, then
 - a. Perform ? Call(*resolvingFunctions*.[[Reject]], **undefined**, « *completion*.[[Value]] »).
11. Return *promise*.

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*.

27.2.4 Properties of the Promise Constructor

The Promise **constructor**:

- has a [[Prototype]] internal slot whose value is %Function.prototype%.
- has the following properties:

27.2.4.1 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 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. Let *promiseResolve* be GetPromiseResolve(*C*).
4. IfAbruptRejectPromise(*promiseResolve*, *promiseCapability*).
5. Let *iteratorRecord* be GetIterator(*iterable*).
6. IfAbruptRejectPromise(*iteratorRecord*, *promiseCapability*).
7. Let *result* be PerformPromiseAll(*iteratorRecord*, *C*, *promiseCapability*, *promiseResolve*).
8. If *result* is an **abrupt completion**, then
 - a. If *iteratorRecord*.[[Done]] is **false**, set *result* to IteratorClose(*iteratorRecord*, *result*).
 - b. IfAbruptRejectPromise(*result*, *promiseCapability*).

9. Return `Completion(result)`.

NOTE The `all` function requires its `this` value to be a `constructor` function that supports the parameter conventions of the `Promise constructor`.

27.2.4.1.1 GetPromiseResolve (*promiseConstructor*)

The abstract operation `GetPromiseResolve` takes argument *promiseConstructor*. It performs the following steps when called:

1. **Assert: IsConstructor**(*promiseConstructor*) is **true**.
2. Let *promiseResolve* be ? `Get`(*promiseConstructor*, "resolve").
3. If `IsCallable`(*promiseResolve*) is **false**, throw a **TypeError** exception.
4. Return *promiseResolve*.

27.2.4.1.2 PerformPromiseAll (*iteratorRecord*, *constructor*, *resultCapability*, *promiseResolve*)

The abstract operation `PerformPromiseAll` takes arguments *iteratorRecord*, *constructor*, *resultCapability* (a `PromiseCapability Record`), and *promiseResolve*. It performs the following steps when called:

1. **Assert: IsConstructor**(*constructor*) is **true**.
2. **Assert: IsCallable**(*promiseResolve*) is **true**.
3. Let *values* be a new empty `List`.
4. Let *remainingElementsCount* be the `Record` { `[[Value]]: 1` }.
5. Let *index* be 0.
6. Repeat,
 - a. Let *next* be `IteratorStep`(*iteratorRecord*).
 - b. If *next* is an **abrupt completion**, set *iteratorRecord*.`[[Done]]` to **true**.
 - c. `ReturnIfAbrupt`(*next*).
 - d. If *next* is **false**, then
 - i. Set *iteratorRecord*.`[[Done]]` to **true**.
 - ii. Set *remainingElementsCount*.`[[Value]]` to *remainingElementsCount*.`[[Value]]` - 1.
 - iii. If *remainingElementsCount*.`[[Value]]` is 0, then
 1. Let *valuesArray* be ! `CreateArrayFromList`(*values*).
 2. Perform ? `Call`(*resultCapability*.`[[Resolve]]`, **undefined**, « *valuesArray* »).
 - iv. Return *resultCapability*.`[[Promise]]`.
 - e. Let *nextValue* be `IteratorValue`(*next*).
 - f. If *nextValue* is an **abrupt completion**, set *iteratorRecord*.`[[Done]]` to **true**.
 - g. `ReturnIfAbrupt`(*nextValue*).
 - h. Append **undefined** to *values*.
 - i. Let *nextPromise* be ? `Call`(*promiseResolve*, *constructor*, « *nextValue* »).
 - j. Let *steps* be the algorithm steps defined in **Promise.all** `Resolve Element Functions`.
 - k. Let *length* be the number of non-optional parameters of the function definition in **Promise.all** `Resolve Element Functions`.
 1. Let *onFulfilled* be ! `CreateBuiltinFunction`(*steps*, *length*, "", « `[[AlreadyCalled]]`, `[[Index]]`, `[[Values]]`, `[[Capability]]`, `[[RemainingElements]]` »).
 - m. Set *onFulfilled*.`[[AlreadyCalled]]` to **false**.
 - n. Set *onFulfilled*.`[[Index]]` to *index*.

- o. Set *onFulfilled*.[[Values]] to *values*.
- p. Set *onFulfilled*.[[Capability]] to *resultCapability*.
- q. Set *onFulfilled*.[[RemainingElements]] to *remainingElementsCount*.
- r. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
- s. Perform ? *Invoke*(*nextPromise*, "then", « *onFulfilled*, *resultCapability*.[[Reject]] »).
- t. Set *index* to *index* + 1.

27.2.4.1.3 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]], [[Capability]], [[RemainingElements]], and [[AlreadyCalled]] internal slots.

When a **Promise.all** resolve element function is called with argument *x*, the following steps are taken:

1. Let *F* be the [active function object](#).
2. If *F*.[[AlreadyCalled]] is **true**, return **undefined**.
3. Set *F*.[[AlreadyCalled]] to **true**.
4. Let *index* be *F*.[[Index]].
5. Let *values* be *F*.[[Values]].
6. Let *promiseCapability* be *F*.[[Capability]].
7. Let *remainingElementsCount* be *F*.[[RemainingElements]].
8. Set *values*[*index*] to *x*.
9. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
10. If *remainingElementsCount*.[[Value]] is 0, then
 - a. Let *valuesArray* be ! [CreateArrayFromList](#)(*values*).
 - b. Return ? [Call](#)(*promiseCapability*.[[Resolve]], **undefined**, « *valuesArray* »).
11. Return **undefined**.

The "length" property of a **Promise.all** resolve element function is 1_F.

27.2.4.2 Promise.allSettled (*iterable*)

The **allSettled** function returns a promise that is fulfilled with an array of promise state snapshots, but only after all the original promises have settled, i.e. become either fulfilled or rejected. 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. Let *promiseResolve* be [GetPromiseResolve](#)(*C*).
4. [IfAbruptRejectPromise](#)(*promiseResolve*, *promiseCapability*).
5. Let *iteratorRecord* be [GetIterator](#)(*iterable*).
6. [IfAbruptRejectPromise](#)(*iteratorRecord*, *promiseCapability*).
7. Let *result* be [PerformPromiseAllSettled](#)(*iteratorRecord*, *C*, *promiseCapability*, *promiseResolve*).
8. If *result* is an [abrupt completion](#), then
 - a. If *iteratorRecord*.[[Done]] is **false**, set *result* to [IteratorClose](#)(*iteratorRecord*, *result*).
 - b. [IfAbruptRejectPromise](#)(*result*, *promiseCapability*).
9. Return [Completion](#)(*result*).

The **allSettled** function requires its **this** value to be a [constructor](#) function that supports the parameter conventions of the Promise [constructor](#).

27.2.4.2.1 PerformPromiseAllSettled (*iteratorRecord*, *constructor*, *resultCapability*, *promiseResolve*)

The abstract operation PerformPromiseAllSettled takes arguments *iteratorRecord*, *constructor*, *resultCapability* (a [PromiseCapability Record](#)), and *promiseResolve*. It performs the following steps when called:

1. Assert: ! [IsConstructor](#)(*constructor*) is **true**.
2. Assert: [IsCallable](#)(*promiseResolve*) is **true**.
3. Let *values* be a new empty [List](#).
4. Let *remainingElementsCount* be the [Record](#) { [\[\[Value\]\]](#): 1 }.
5. Let *index* be 0.
6. Repeat,
 - a. Let *next* be [IteratorStep](#)(*iteratorRecord*).
 - b. If *next* is an [abrupt completion](#), set *iteratorRecord*.[\[\[Done\]\]](#) to **true**.
 - c. [ReturnIfAbrupt](#)(*next*).
 - d. If *next* is **false**, then
 - i. Set *iteratorRecord*.[\[\[Done\]\]](#) to **true**.
 - ii. Set *remainingElementsCount*.[\[\[Value\]\]](#) to *remainingElementsCount*.[\[\[Value\]\]](#) - 1.
 - iii. If *remainingElementsCount*.[\[\[Value\]\]](#) is 0, then
 1. Let *valuesArray* be ! [CreateArrayFromList](#)(*values*).
 2. Perform ? [Call](#)(*resultCapability*.[\[\[Resolve\]\]](#), **undefined**, « *valuesArray* »).
 - iv. Return *resultCapability*.[\[\[Promise\]\]](#).
 - e. Let *nextValue* be [IteratorValue](#)(*next*).
 - f. If *nextValue* is an [abrupt completion](#), set *iteratorRecord*.[\[\[Done\]\]](#) to **true**.
 - g. [ReturnIfAbrupt](#)(*nextValue*).
 - h. Append **undefined** to *values*.
 - i. Let *nextPromise* be ? [Call](#)(*promiseResolve*, *constructor*, « *nextValue* »).
 - j. Let *stepsFulfilled* be the algorithm steps defined in [Promise.allSettled Resolve Element Functions](#).
 - k. Let *lengthFulfilled* be the number of non-optional parameters of the function definition in [Promise.allSettled Resolve Element Functions](#).
 - l. Let *onFulfilled* be ! [CreateBuiltinFunction](#)(*stepsFulfilled*, *lengthFulfilled*, "", « [\[\[AlreadyCalled\]\]](#), [\[\[Index\]\]](#), [\[\[Values\]\]](#), [\[\[Capability\]\]](#), [\[\[RemainingElements\]\]](#) »).
 - m. Let *alreadyCalled* be the [Record](#) { [\[\[Value\]\]](#): **false** }.
 - n. Set *onFulfilled*.[\[\[AlreadyCalled\]\]](#) to *alreadyCalled*.
 - o. Set *onFulfilled*.[\[\[Index\]\]](#) to *index*.
 - p. Set *onFulfilled*.[\[\[Values\]\]](#) to *values*.
 - q. Set *onFulfilled*.[\[\[Capability\]\]](#) to *resultCapability*.
 - r. Set *onFulfilled*.[\[\[RemainingElements\]\]](#) to *remainingElementsCount*.
 - s. Let *stepsRejected* be the algorithm steps defined in [Promise.allSettled Reject Element Functions](#).
 - t. Let *lengthRejected* be the number of non-optional parameters of the function definition in [Promise.allSettled Reject Element Functions](#).
 - u. Let *onRejected* be ! [CreateBuiltinFunction](#)(*stepsRejected*, *lengthRejected*, "", « [\[\[AlreadyCalled\]\]](#), [\[\[Index\]\]](#), [\[\[Values\]\]](#), [\[\[Capability\]\]](#), [\[\[RemainingElements\]\]](#) »).
 - v. Set *onRejected*.[\[\[AlreadyCalled\]\]](#) to *alreadyCalled*.

- w. Set *onRejected*.[[Index]] to *index*.
- x. Set *onRejected*.[[Values]] to *values*.
- y. Set *onRejected*.[[Capability]] to *resultCapability*.
- z. Set *onRejected*.[[RemainingElements]] to *remainingElementsCount*.
- aa. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
- ab. Perform ? *Invoke*(*nextPromise*, "then", « *onFulfilled*, *onRejected* »).
- ac. Set *index* to *index* + 1.

27.2.4.2.2 **Promise.allSettled** Resolve Element Functions

A **Promise.allSettled** resolve element function is an anonymous built-in function that is used to resolve a specific **Promise.allSettled** element. Each **Promise.allSettled** resolve element function has [[Index]], [[Values]], [[Capability]], [[RemainingElements]], and [[AlreadyCalled]] internal slots.

When a **Promise.allSettled** resolve element function is called with argument *x*, the following steps are taken:

1. Let *F* be the **active function object**.
2. Let *alreadyCalled* be *F*.[[AlreadyCalled]].
3. If *alreadyCalled*.[[Value]] is **true**, return **undefined**.
4. Set *alreadyCalled*.[[Value]] to **true**.
5. Let *index* be *F*.[[Index]].
6. Let *values* be *F*.[[Values]].
7. Let *promiseCapability* be *F*.[[Capability]].
8. Let *remainingElementsCount* be *F*.[[RemainingElements]].
9. Let *obj* be ! **OrdinaryObjectCreate**(%Object.prototype%).
10. Perform ! **CreateDataPropertyOrThrow**(*obj*, "status", "fulfilled").
11. Perform ! **CreateDataPropertyOrThrow**(*obj*, "value", *x*).
12. Set *values*[*index*] to *obj*.
13. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
14. If *remainingElementsCount*.[[Value]] is 0, then
 - a. Let *valuesArray* be ! **CreateArrayFromList**(*values*).
 - b. Return ? **Call**(*promiseCapability*.[[Resolve]], **undefined**, « *valuesArray* »).
15. Return **undefined**.

The "length" property of a **Promise.allSettled** resolve element function is 1_F.

27.2.4.2.3 **Promise.allSettled** Reject Element Functions

A **Promise.allSettled** reject element function is an anonymous built-in function that is used to reject a specific **Promise.allSettled** element. Each **Promise.allSettled** reject element function has [[Index]], [[Values]], [[Capability]], [[RemainingElements]], and [[AlreadyCalled]] internal slots.

When a **Promise.allSettled** reject element function is called with argument *x*, the following steps are taken:

1. Let *F* be the **active function object**.
2. Let *alreadyCalled* be *F*.[[AlreadyCalled]].
3. If *alreadyCalled*.[[Value]] is **true**, return **undefined**.
4. Set *alreadyCalled*.[[Value]] to **true**.
5. Let *index* be *F*.[[Index]].

6. Let *values* be *F*.[[Values]].
7. Let *promiseCapability* be *F*.[[Capability]].
8. Let *remainingElementsCount* be *F*.[[RemainingElements]].
9. Let *obj* be ! OrdinaryObjectCreate(%Object.prototype%).
10. Perform ! CreateDataPropertyOrThrow(*obj*, "status", "rejected").
11. Perform ! CreateDataPropertyOrThrow(*obj*, "reason", *x*).
12. Set *values*[*index*] to *obj*.
13. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
14. If *remainingElementsCount*.[[Value]] is 0, then
 - a. Let *valuesArray* be ! CreateArrayFromList(*values*).
 - b. Return ? Call(*promiseCapability*.[[Resolve]], undefined, « *valuesArray* »).
15. Return undefined.

The "length" property of a **Promise.allSettled** reject element function is 1_F.

27.2.4.3 Promise.any (*iterable*)

The **any** function returns a promise that is fulfilled by the first given promise to be fulfilled, or rejected with an **AggregateError** holding the rejection reasons if all of the given promises are rejected. 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. Let *promiseResolve* be GetPromiseResolve(*C*).
4. IfAbruptRejectPromise(*promiseResolve*, *promiseCapability*).
5. Let *iteratorRecord* be GetIterator(*iterable*).
6. IfAbruptRejectPromise(*iteratorRecord*, *promiseCapability*).
7. Let *result* be PerformPromiseAny(*iteratorRecord*, *C*, *promiseCapability*, *promiseResolve*).
8. If *result* is an abrupt completion, then
 - a. If *iteratorRecord*.[[Done]] is **false**, set *result* to IteratorClose(*iteratorRecord*, *result*).
 - b. IfAbruptRejectPromise(*result*, *promiseCapability*).
9. Return Completion(*result*).

NOTE The **any** function requires its **this** value to be a **constructor** function that supports the parameter conventions of the **Promise constructor**.

27.2.4.3.1 PerformPromiseAny (*iteratorRecord*, *constructor*, *resultCapability*, *promiseResolve*)

The abstract operation PerformPromiseAny takes arguments *iteratorRecord*, *constructor*, *resultCapability* (a **PromiseCapability Record**), and *promiseResolve*. It performs the following steps when called:

1. Assert: ! IsConstructor(*constructor*) is **true**.
2. Assert: ! IsCallable(*promiseResolve*) is **true**.
3. Let *errors* be a new empty List.
4. Let *remainingElementsCount* be the Record { [[Value]]: 1 }.
5. Let *index* be 0.
6. Repeat,
 - a. Let *next* be IteratorStep(*iteratorRecord*).

- b. If *next* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
- c. **ReturnIfAbrupt**(*next*).
- d. If *next* is **false**, then
 - i. Set *iteratorRecord*.[[Done]] to **true**.
 - ii. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
 - iii. If *remainingElementsCount*.[[Value]] is 0, then
 1. Let *error* be a newly created **AggregateError** object.
 2. Perform ! **DefinePropertyOrThrow**(*error*, "errors", PropertyDescriptor { [[Configurable]]: **true**, [[Enumerable]]: **false**, [[Writable]]: **true**, [[Value]]: ! **CreateArrayFromList**(*errors*) }).
 3. Return **ThrowCompletion**(*error*).
 - iv. Return *resultCapability*.[[Promise]].
- e. Let *nextValue* be **IteratorValue**(*next*).
- f. If *nextValue* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.
- g. **ReturnIfAbrupt**(*nextValue*).
- h. Append **undefined** to *errors*.
- i. Let *nextPromise* be ? **Call**(*promiseResolve*, *constructor*, « *nextValue* »).
- j. Let *stepsRejected* be the algorithm steps defined in **Promise.any** Reject Element Functions.
- k. Let *lengthRejected* be the number of non-optional parameters of the function definition in **Promise.any** Reject Element Functions.
 1. Let *onRejected* be ! **CreateBuiltinFunction**(*stepsRejected*, *lengthRejected*, "", « [[AlreadyCalled]], [[Index]], [[Errors]], [[Capability]], [[RemainingElements]] »).
- m. Set *onRejected*.[[AlreadyCalled]] to **false**.
- n. Set *onRejected*.[[Index]] to *index*.
- o. Set *onRejected*.[[Errors]] to *errors*.
- p. Set *onRejected*.[[Capability]] to *resultCapability*.
- q. Set *onRejected*.[[RemainingElements]] to *remainingElementsCount*.
- r. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] + 1.
- s. Perform ? **Invoke**(*nextPromise*, "then", « *resultCapability*.[[Resolve]], *onRejected* »).
- t. Set *index* to *index* + 1.

27.2.4.3.2 **Promise.any** Reject Element Functions

A **Promise.any** reject element function is an anonymous built-in function that is used to reject a specific **Promise.any** element. Each **Promise.any** reject element function has [[Index]], [[Errors]], [[Capability]], [[RemainingElements]], and [[AlreadyCalled]] internal slots.

When a **Promise.any** reject element function is called with argument *x*, the following steps are taken:

1. Let *F* be the **active function object**.
2. If *F*.[[AlreadyCalled]] is **true**, return **undefined**.
3. Set *F*.[[AlreadyCalled]] to **true**.
4. Let *index* be *F*.[[Index]].
5. Let *errors* be *F*.[[Errors]].
6. Let *promiseCapability* be *F*.[[Capability]].
7. Let *remainingElementsCount* be *F*.[[RemainingElements]].
8. Set *errors*[*index*] to *x*.
9. Set *remainingElementsCount*.[[Value]] to *remainingElementsCount*.[[Value]] - 1.
10. If *remainingElementsCount*.[[Value]] is 0, then

- a. Let *error* be a newly created **AggregateError** object.
 - b. Perform **! DefinePropertyOrThrow**(*error*, "errors", PropertyDescriptor { [[Configurable]]: **true**, [[Enumerable]]: **false**, [[Writable]]: **true**, [[Value]]: **! CreateArrayFromList**(*errors*) }).
 - c. Return **? Call**(*promiseCapability*.[[Reject]], **undefined**, « *error* »).
11. Return **undefined**.

The "length" property of a **Promise.any** reject element function is 1_F.

27.2.4.4 Promise.prototype

The initial value of **Promise.prototype** is the [Promise prototype object](#).

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

27.2.4.5 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. Let *promiseResolve* be **GetPromiseResolve**(*C*).
4. **IfAbruptRejectPromise**(*promiseResolve*, *promiseCapability*).
5. Let *iteratorRecord* be **GetIterator**(*iterable*).
6. **IfAbruptRejectPromise**(*iteratorRecord*, *promiseCapability*).
7. Let *result* be **PerformPromiseRace**(*iteratorRecord*, *C*, *promiseCapability*, *promiseResolve*).
8. If *result* is an **abrupt completion**, then
 - a. If *iteratorRecord*.[[Done]] is **false**, set *result* to **IteratorClose**(*iteratorRecord*, *result*).
 - b. **IfAbruptRejectPromise**(*result*, *promiseCapability*).
9. Return **Completion**(*result*).

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.

27.2.4.5.1 PerformPromiseRace (*iteratorRecord*, *constructor*, *resultCapability*, *promiseResolve*)

The abstract operation **PerformPromiseRace** takes arguments *iteratorRecord*, *constructor*, *resultCapability* (a **PromiseCapability Record**), and *promiseResolve*. It performs the following steps when called:

1. **Assert: IsConstructor**(*constructor*) is **true**.
2. **Assert: IsCallable**(*promiseResolve*) is **true**.
3. Repeat,
 - a. Let *next* be **IteratorStep**(*iteratorRecord*).
 - b. If *next* is an **abrupt completion**, set *iteratorRecord*.[[Done]] to **true**.

- c. `ReturnIfAbrupt(next)`.
- d. If `next` is `false`, then
 - i. Set `iteratorRecord.[[Done]]` to `true`.
 - ii. Return `resultCapability.[[Promise]]`.
- e. Let `nextValue` be `IteratorValue(next)`.
- f. If `nextValue` is an `abrupt completion`, set `iteratorRecord.[[Done]]` to `true`.
- g. `ReturnIfAbrupt(nextValue)`.
- h. Let `nextPromise` be `? Call(promiseResolve, constructor, « nextValue »)`.
- i. Perform `? Invoke(nextPromise, "then", « resultCapability.[[Resolve]], resultCapability.[[Reject]] »)`.

27.2.4.6 `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. Perform `? Call(promiseCapability.[[Reject]], undefined, « r »)`.
4. Return `promiseCapability.[[Promise]]`.

NOTE The **reject** function expects its **this** value to be a `constructor` function that supports the parameter conventions of the `Promise constructor`.

27.2.4.7 `Promise.resolve (x)`

The **resolve** function returns either a new promise resolved with the passed argument, or the argument itself if the argument is a promise produced by this `constructor`.

1. Let `C` be the **this** value.
2. If `Type(C)` is not `Object`, throw a `TypeError` exception.
3. Return `? PromiseResolve(C, x)`.

NOTE The **resolve** function expects its **this** value to be a `constructor` function that supports the parameter conventions of the `Promise constructor`.

27.2.4.7.1 `PromiseResolve (C, x)`

The abstract operation `PromiseResolve` takes arguments `C` (a `constructor`) and `x` (an `ECMAScript language value`). It returns a new promise resolved with `x`. It performs the following steps when called:

1. **Assert:** `Type(C)` is `Object`.
2. If `IsPromise(x)` is `true`, then
 - a. Let `xConstructor` be `? Get(x, "constructor")`.
 - b. If `SameValue(xConstructor, C)` is `true`, return `x`.
3. Let `promiseCapability` be `? NewPromiseCapability(C)`.
4. Perform `? Call(promiseCapability.[[Resolve]], undefined, « x »)`.
5. Return `promiseCapability.[[Promise]]`.

27.2.4.8 get Promise [@@species]

Promise[@@species] is an [accessor property](#) whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Return the **this** value.

The value of the "name" property of this function is "get [Symbol.species]".

NOTE Promise prototype methods normally use their **this** value's [constructor](#) to create a derived object. However, a subclass [constructor](#) may over-ride that default behaviour by redefining its [@@species](#) property.

27.2.5 Properties of the Promise Prototype Object

The *Promise prototype object*:

- is `%Promise.prototype%`.
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is an [ordinary object](#).
- does not have a `[[PromiseState]]` internal slot or any of the other internal slots of Promise instances.

27.2.5.1 Promise.prototype.catch (*onRejected*)

When the **catch** method is called with argument *onRejected*, the following steps are taken:

1. Let *promise* be the **this** value.
2. Return ? [Invoke](#)(*promise*, "then", « **undefined**, *onRejected* »).

27.2.5.2 Promise.prototype.constructor

The initial value of **Promise.prototype.constructor** is `%Promise%`.

27.2.5.3 Promise.prototype.finally (*onFinally*)

When the **finally** method is called with argument *onFinally*, the following steps are taken:

1. Let *promise* be the **this** value.
2. If [Type](#)(*promise*) is not Object, throw a **TypeError** exception.
3. Let *C* be ? [SpeciesConstructor](#)(*promise*, `%Promise%`).
4. **Assert**: [IsConstructor](#)(*C*) is **true**.
5. If [IsCallable](#)(*onFinally*) is **false**, then
 - a. Let *thenFinally* be *onFinally*.
 - b. Let *catchFinally* be *onFinally*.
6. Else,
 - a. Let *stepsThenFinally* be the algorithm steps defined in [Then Finally Functions](#).
 - b. Let *lengthThenFinally* be the number of non-optional parameters of the function definition in [Then Finally Functions](#).
 - c. Let *thenFinally* be ! [CreateBuiltinFunction](#)(*stepsThenFinally*, *lengthThenFinally*, "", « `[[Constructor]]`),

- [[OnFinally]] »).
 - d. Set *thenFinally*.[[Constructor]] to *C*.
 - e. Set *thenFinally*.[[OnFinally]] to *onFinally*.
 - f. Let *stepsCatchFinally* be the algorithm steps defined in [Catch Finally Functions](#).
 - g. Let *lengthCatchFinally* be the number of non-optional parameters of the function definition in [Catch Finally Functions](#).
 - h. Let *catchFinally* be ! [CreateBuiltinFunction](#)(*stepsCatchFinally*, *lengthCatchFinally*, "", « [[Constructor]], [[OnFinally]] »).
 - i. Set *catchFinally*.[[Constructor]] to *C*.
 - j. Set *catchFinally*.[[OnFinally]] to *onFinally*.
7. Return ? [Invoke](#)(*promise*, "then", « *thenFinally*, *catchFinally* »).

27.2.5.3.1 Then Finally Functions

A Then Finally function is an anonymous built-in function that has a [[Constructor]] and an [[OnFinally]] internal slot. The value of the [[Constructor]] internal slot is a **Promise**-like [constructor function object](#), and the value of the [[OnFinally]] internal slot is a [function object](#).

When a Then Finally function is called with argument *value*, the following steps are taken:

1. Let *F* be the [active function object](#).
2. Let *onFinally* be *F*.[[OnFinally]].
3. **Assert:** [IsCallable](#)(*onFinally*) is **true**.
4. Let *result* be ? [Call](#)(*onFinally*, **undefined**).
5. Let *C* be *F*.[[Constructor]].
6. **Assert:** [IsConstructor](#)(*C*) is **true**.
7. Let *promise* be ? [PromiseResolve](#)(*C*, *result*).
8. Let *valueThink* be equivalent to a function that returns *value*.
9. Return ? [Invoke](#)(*promise*, "then", « *valueThink* »).

The "length" property of a Then Finally function is 1_F.

27.2.5.3.2 Catch Finally Functions

A Catch Finally function is an anonymous built-in function that has a [[Constructor]] and an [[OnFinally]] internal slot. The value of the [[Constructor]] internal slot is a **Promise**-like [constructor function object](#), and the value of the [[OnFinally]] internal slot is a [function object](#).

When a Catch Finally function is called with argument *reason*, the following steps are taken:

1. Let *F* be the [active function object](#).
2. Let *onFinally* be *F*.[[OnFinally]].
3. **Assert:** [IsCallable](#)(*onFinally*) is **true**.
4. Let *result* be ? [Call](#)(*onFinally*, **undefined**).
5. Let *C* be *F*.[[Constructor]].
6. **Assert:** [IsConstructor](#)(*C*) is **true**.
7. Let *promise* be ? [PromiseResolve](#)(*C*, *result*).
8. Let *thrower* be equivalent to a function that throws *reason*.
9. Return ? [Invoke](#)(*promise*, "then", « *thrower* »).

The "length" property of a Catch Finally function is 1.

27.2.5.4 Promise.prototype.then (*onFulfilled*, *onRejected*)

When the **then** method is called with arguments *onFulfilled* and *onRejected*, the following steps are taken:

1. Let *promise* be the **this** value.
2. If **IsPromise**(*promise*) is **false**, throw a **TypeError** exception.
3. Let *C* be ? **SpeciesConstructor**(*promise*, %**Promise**%).
4. Let *resultCapability* be ? **NewPromiseCapability**(*C*).
5. Return **PerformPromiseThen**(*promise*, *onFulfilled*, *onRejected*, *resultCapability*).

27.2.5.4.1 PerformPromiseThen (*promise*, *onFulfilled*, *onRejected* [, *resultCapability*])

The abstract operation **PerformPromiseThen** takes arguments *promise*, *onFulfilled*, and *onRejected* and optional argument *resultCapability* (a **PromiseCapability Record**). It performs the "then" operation on *promise* using *onFulfilled* and *onRejected* as its settlement actions. If *resultCapability* is passed, the result is stored by updating *resultCapability*'s promise. If it is not passed, then **PerformPromiseThen** is being called by a specification-internal operation where the result does not matter. It performs the following steps when called:

1. **Assert**: **IsPromise**(*promise*) is **true**.
2. If *resultCapability* is not present, then
 - a. Set *resultCapability* to **undefined**.
3. If **IsCallable**(*onFulfilled*) is **false**, then
 - a. Let *onFulfilledJobCallback* be empty.
4. Else,
 - a. Let *onFulfilledJobCallback* be **HostMakeJobCallback**(*onFulfilled*).
5. If **IsCallable**(*onRejected*) is **false**, then
 - a. Let *onRejectedJobCallback* be empty.
6. Else,
 - a. Let *onRejectedJobCallback* be **HostMakeJobCallback**(*onRejected*).
7. Let *fulfillReaction* be the **PromiseReaction** { **[[Capability]]**: *resultCapability*, **[[Type]]**: **Fulfill**, **[[Handler]]**: *onFulfilledJobCallback* }.
8. Let *rejectReaction* be the **PromiseReaction** { **[[Capability]]**: *resultCapability*, **[[Type]]**: **Reject**, **[[Handler]]**: *onRejectedJobCallback* }.
9. If *promise*.**[[PromiseState]]** is pending, then
 - a. Append *fulfillReaction* as the last element of the **List** that is *promise*.**[[PromiseFulfillReactions]]**.
 - b. Append *rejectReaction* as the last element of the **List** that is *promise*.**[[PromiseRejectReactions]]**.
10. Else if *promise*.**[[PromiseState]]** is fulfilled, then
 - a. Let *value* be *promise*.**[[PromiseResult]]**.
 - b. Let *fulfillJob* be **NewPromiseReactionJob**(*fulfillReaction*, *value*).
 - c. Perform **HostEnqueuePromiseJob**(*fulfillJob*.**[[Job]]**, *fulfillJob*.**[[Realm]]**).
11. Else,
 - a. **Assert**: The value of *promise*.**[[PromiseState]]** is rejected.
 - b. Let *reason* be *promise*.**[[PromiseResult]]**.
 - c. If *promise*.**[[PromiseIsHandled]]** is **false**, perform **HostPromiseRejectionTracker**(*promise*, "handle").
 - d. Let *rejectJob* be **NewPromiseReactionJob**(*rejectReaction*, *reason*).
 - e. Perform **HostEnqueuePromiseJob**(*rejectJob*.**[[Job]]**, *rejectJob*.**[[Realm]]**).

12. Set *promise*.[[PromiseIsHandled]] to **true**.
13. If *resultCapability* is **undefined**, then
 - a. Return **undefined**.
14. Else,
 - a. Return *resultCapability*.[[Promise]].

27.2.5.5 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** }.

27.2.6 Properties of Promise Instances

Promise instances are ordinary objects that inherit properties from the [Promise prototype object](#) (the intrinsic, `%Promise.prototype%`). Promise instances are initially created with the internal slots described in [Table 72](#).

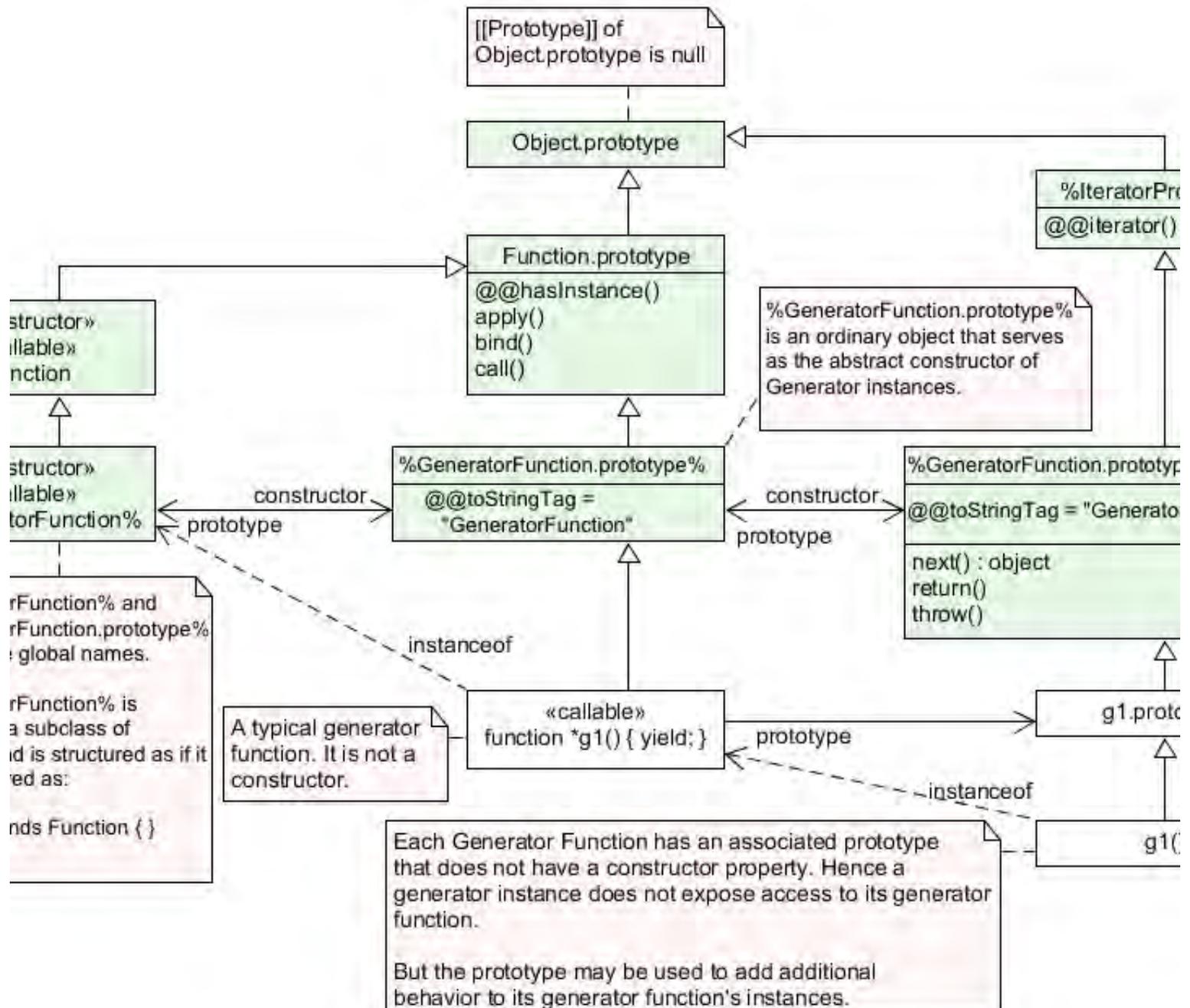
Table 72: Internal Slots of Promise Instances

Internal Slot	Description
[[PromiseState]]	One of pending , fulfilled , or rejected . Governs how a promise will react to incoming calls to its then method.
[[PromiseResult]]	The value with which the promise has been fulfilled or rejected, if any. Only meaningful if [[PromiseState]] is not pending .
[[PromiseFulfillReactions]]	A List of PromiseReaction records to be processed when/if the promise transitions from the pending state to the fulfilled state.
[[PromiseRejectReactions]]	A List of PromiseReaction records to be processed when/if the promise transitions from the pending state to the rejected state.
[[PromiseIsHandled]]	A boolean indicating whether the promise has ever had a fulfillment or rejection handler; used in unhandled rejection tracking.

27.3 GeneratorFunction Objects

GeneratorFunction objects are functions that are usually created by evaluating *GeneratorDeclarations*, *GeneratorExpressions*, and *GeneratorMethods*. They may also be created by calling the `%GeneratorFunction%` intrinsic.

Figure 5 (Informative): Generator Objects Relationships



27.3.1 The GeneratorFunction Constructor

The GeneratorFunction [constructor](#):

- is `%GeneratorFunction%`.
- is a subclass of **Function**.
- creates and initializes a new GeneratorFunction object when called as a function rather than as a [constructor](#). Thus the function call **GeneratorFunction (...)** is equivalent to the object creation expression **new GeneratorFunction (...)** with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified GeneratorFunction behaviour must include a **super** call to the

GeneratorFunction [constructor](#) to create and initialize subclass instances with the internal slots necessary for built-in GeneratorFunction behaviour. All ECMAScript syntactic forms for defining generator function objects create direct instances of GeneratorFunction. There is no syntactic means to create instances of GeneratorFunction subclasses.

27.3.1.1 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 *C* be the [active function object](#).
2. Let *args* be the [argumentsList](#) that was passed to this function by `[[Call]]` or `[[Construct]]`.
3. Return ? [CreateDynamicFunction](#)(*C*, *NewTarget*, *generator*, *args*).

NOTE See NOTE for [20.2.1.1](#).

27.3.2 Properties of the GeneratorFunction Constructor

The GeneratorFunction [constructor](#):

- is a standard built-in [function object](#) that inherits from the Function [constructor](#).
- has a `[[Prototype]]` internal slot whose value is `%Function%`.
- has a "name" property whose value is "GeneratorFunction".
- has the following properties:

27.3.2.1 GeneratorFunction.length

This is a [data property](#) with a value of 1. This property has the attributes { `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **true** }.

27.3.2.2 GeneratorFunction.prototype

The initial value of **GeneratorFunction.prototype** is the [GeneratorFunction prototype object](#).

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

27.3.3 Properties of the GeneratorFunction Prototype Object

The *GeneratorFunction prototype object*:

- is `%GeneratorFunction.prototype%` (see [Figure 5](#)).
- is an [ordinary object](#).
- is not a [function object](#) and does not have an `[[ECMAScriptCode]]` internal slot or any other of the internal slots listed in [Table 29](#) or [Table 73](#).
- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.

27.3.3.1 GeneratorFunction.prototype.constructor

The initial value of **GeneratorFunction.prototype.constructor** is `%GeneratorFunction%`.

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

27.3.3.2 GeneratorFunction.prototype.prototype

The initial value of **GeneratorFunction.prototype.prototype** is the [Generator prototype object](#).

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

27.3.3.3 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` }.

27.3.4 GeneratorFunction Instances

Every GeneratorFunction instance is an ECMAScript [function object](#) and has the internal slots listed in [Table 29](#). The value of the `[[IsClassConstructor]]` internal slot for all such instances is `false`.

Each GeneratorFunction instance has the following own properties:

27.3.4.1 length

The specification for the `"length"` property of Function instances given in [20.2.4.1](#) also applies to GeneratorFunction instances.

27.3.4.2 name

The specification for the `"name"` property of Function instances given in [20.2.4.2](#) also applies to GeneratorFunction instances.

27.3.4.3 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 when the generator [function object](#) is invoked using `[[Call]]`.

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

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.

27.4 AsyncGeneratorFunction Objects

AsyncGeneratorFunction objects are functions that are usually created by evaluating *AsyncGeneratorDeclaration*,

AsyncGeneratorExpression, and *AsyncGeneratorMethod* syntactic productions. They may also be created by calling the `%AsyncGeneratorFunction%` intrinsic.

27.4.1 The AsyncGeneratorFunction Constructor

The AsyncGeneratorFunction [constructor](#):

- is `%AsyncGeneratorFunction%`.
- is a subclass of **Function**.
- creates and initializes a new AsyncGeneratorFunction object when called as a function rather than as a [constructor](#). Thus the function call **AsyncGeneratorFunction (...)** is equivalent to the object creation expression **new AsyncGeneratorFunction (...)** with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified AsyncGeneratorFunction behaviour must include a **super** call to the AsyncGeneratorFunction [constructor](#) to create and initialize subclass instances with the internal slots necessary for built-in AsyncGeneratorFunction behaviour. All ECMAScript syntactic forms for defining async generator function objects create direct instances of AsyncGeneratorFunction. There is no syntactic means to create instances of AsyncGeneratorFunction subclasses.

27.4.1.1 AsyncGeneratorFunction (*p1*, *p2*, ... , *pn*, *body*)

The last argument specifies the body (executable code) of an async generator function; any preceding arguments specify formal parameters.

When the **AsyncGeneratorFunction** 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 *C* be the [active function object](#).
2. Let *args* be the [argumentsList](#) that was passed to this function by `[[Call]]` or `[[Construct]]`.
3. Return ? [CreateDynamicFunction](#)(*C*, `NewTarget`, `asyncGenerator`, *args*).

NOTE See NOTE for [20.2.1.1](#).

27.4.2 Properties of the AsyncGeneratorFunction Constructor

The AsyncGeneratorFunction [constructor](#):

- is a standard built-in [function object](#) that inherits from the Function [constructor](#).
- has a `[[Prototype]]` internal slot whose value is `%Function%`.
- has a `"name"` property whose value is `"AsyncGeneratorFunction"`.
- has the following properties:

27.4.2.1 AsyncGeneratorFunction.length

This is a [data property](#) with a value of 1. This property has the attributes { `[[Writable]]`: **false**, `[[Enumerable]]`: **false**, `[[Configurable]]`: **true** }.

27.4.2.2 AsyncGeneratorFunction.prototype

The initial value of **AsyncGeneratorFunction.prototype** is the [AsyncGeneratorFunction prototype object](#).

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

27.4.3 Properties of the AsyncGeneratorFunction Prototype Object

The *AsyncGeneratorFunction prototype object*:

- is *%AsyncGeneratorFunction.prototype%*.
- is an [ordinary object](#).
- is not a [function object](#) and does not have an **[[ECMAScriptCode]]** internal slot or any other of the internal slots listed in [Table 29](#) or [Table 74](#).
- has a **[[Prototype]]** internal slot whose value is *%Function.prototype%*.

27.4.3.1 AsyncGeneratorFunction.prototype.constructor

The initial value of **AsyncGeneratorFunction.prototype.constructor** is *%AsyncGeneratorFunction%*.

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

27.4.3.2 AsyncGeneratorFunction.prototype.prototype

The initial value of **AsyncGeneratorFunction.prototype.prototype** is the [AsyncGenerator prototype object](#).

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

27.4.3.3 AsyncGeneratorFunction.prototype [@@toStringTag]

The initial value of the **@@toStringTag** property is the String value "AsyncGeneratorFunction".

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

27.4.4 AsyncGeneratorFunction Instances

Every AsyncGeneratorFunction instance is an ECMAScript [function object](#) and has the internal slots listed in [Table 29](#). The value of the **[[IsClassConstructor]]** internal slot for all such instances is **false**.

Each AsyncGeneratorFunction instance has the following own properties:

27.4.4.1 length

The value of the **"length"** property is an [integral Number](#) that indicates the typical number of arguments expected by the AsyncGeneratorFunction. However, the language permits the function to be invoked with some other number of arguments. The behaviour of an AsyncGeneratorFunction 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` }.

27.4.4.2 name

The specification for the "name" property of Function instances given in 20.2.4.2 also applies to AsyncGeneratorFunction instances.

27.4.4.3 prototype

Whenever an AsyncGeneratorFunction instance is created another [ordinary object](#) is also created and is the initial value of the async generator function's "prototype" property. The value of the prototype property is used to initialize the `[[Prototype]]` internal slot of a newly created AsyncGenerator object when the generator [function object](#) is invoked using `[[Call]]`.

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

NOTE Unlike function instances, the object that is the value of the an AsyncGeneratorFunction's "prototype" property does not have a "constructor" property whose value is the AsyncGeneratorFunction instance.

27.5 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 initial value of the "prototype" property of the Generator function that created the instance. Generator instances indirectly inherit properties from the Generator Prototype intrinsic, `%GeneratorFunction.prototype.prototype%`.

27.5.1 Properties of the Generator Prototype Object

The *Generator prototype object*:

- is `%GeneratorFunction.prototype.prototype%`.
- is an [ordinary object](#).
- is not a Generator instance and does not have a `[[GeneratorState]]` internal slot.
- has a `[[Prototype]]` internal slot whose value is `%IteratorPrototype%`.
- has properties that are indirectly inherited by all Generator instances.

27.5.1.1 Generator.prototype.constructor

The initial value of `Generator.prototype.constructor` is `%GeneratorFunction.prototype%`.

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

27.5.1.2 Generator.prototype.next (*value*)

The **next** method performs the following steps:

1. Let *g* be the **this** value.
2. Return ? `GeneratorResume(g, value, empty)`.

27.5.1.3 `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, empty)`.

27.5.1.4 `Generator.prototype.throw (exception)`

The **throw** method performs the following steps:

1. Let *g* be the **this** value.
2. Let *C* be `ThrowCompletion(exception)`.
3. Return ? `GeneratorResumeAbrupt(g, C, empty)`.

27.5.1.5 `Generator.prototype [@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value "Generator".

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

27.5.2 Properties of Generator Instances

Generator instances are initially created with the internal slots described in [Table 73](#).

Table 73: Internal Slots of Generator Instances

Internal Slot	Description
<code>[[GeneratorState]]</code>	The current execution state of the generator. The possible values are: undefined , <code>suspendedStart</code> , <code>suspendedYield</code> , <code>executing</code> , and <code>completed</code> .
<code>[[GeneratorContext]]</code>	The execution context that is used when executing the code of this generator.
<code>[[GeneratorBrand]]</code>	A brand used to distinguish different kinds of generators. The <code>[[GeneratorBrand]]</code> of generators declared by ECMAScript source text is always <code>empty</code> .

27.5.3 Generator Abstract Operations

27.5.3.1 `GeneratorStart (generator, generatorBody)`

The abstract operation `GeneratorStart` takes arguments *generator* and *generatorBody* (a [Parse Node](#) or an [Abstract Closure](#) with no parameters). It performs the following steps when called:

1. **Assert**: The value of *generator*.`[[GeneratorState]]` 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. If *generatorBody* is a *Parse Node*, then
 - i. Let *result* be the result of evaluating *generatorBody*.
 - b. Else,
 - i. **Assert:** *generatorBody* is an *Abstract Closure* with no parameters.
 - ii. Let *result* be *generatorBody*().
 - c. **Assert:** If we return here, the generator either threw an exception or performed either an implicit or explicit return.
 - d. 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*.
 - e. Set *generator*.[[GeneratorState]] to *completed*.
 - f. 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.
 - g. If *result*.[[Type]] is *normal*, let *resultValue* be **undefined**.
 - h. Else if *result*.[[Type]] is *return*, let *resultValue* be *result*.[[Value]].
 - i. Else,
 - i. **Assert:** *result*.[[Type]] is *throw*.
 - ii. Return *Completion*(*result*).
 - j. Return *CreateIterResultObject*(*resultValue*, **true**).
5. Set *generator*.[[GeneratorContext]] to *genContext*.
6. Set *generator*.[[GeneratorState]] to *suspendedStart*.
7. Return *NormalCompletion*(**undefined**).

27.5.3.2 GeneratorValidate (*generator*, *generatorBrand*)

The abstract operation *GeneratorValidate* takes arguments *generator* and *generatorBrand*. It performs the following steps when called:

1. Perform ? *RequireInternalSlot*(*generator*, [[GeneratorState]]).
2. Perform ? *RequireInternalSlot*(*generator*, [[GeneratorBrand]]).
3. If *generator*.[[GeneratorBrand]] is not the same value as *generatorBrand*, throw a **TypeError** exception.
4. **Assert:** *generator* also has a [[GeneratorContext]] internal slot.
5. Let *state* be *generator*.[[GeneratorState]].
6. If *state* is *executing*, throw a **TypeError** exception.
7. Return *state*.

27.5.3.3 GeneratorResume (*generator*, *value*, *generatorBrand*)

The abstract operation *GeneratorResume* takes arguments *generator*, *value*, and *generatorBrand*. It performs the following steps when called:

1. Let *state* be ? *GeneratorValidate*(*generator*, *generatorBrand*).
2. If *state* is *completed*, return *CreateIterResultObject*(**undefined**, **true**).
3. **Assert:** *state* is either *suspendedStart* or *suspendedYield*.
4. Let *genContext* be *generator*.[[GeneratorContext]].
5. Let *methodContext* be the *running execution context*.

6. Suspend *methodContext*.
7. Set *generator*.[[GeneratorState]] to **executing**.
8. Push *genContext* onto the **execution context stack**; *genContext* is now the **running execution context**.
9. 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.
10. **Assert**: When we return here, *genContext* has already been removed from the **execution context stack** and *methodContext* is the currently **running execution context**.
11. Return **Completion(result)**.

27.5.3.4 GeneratorResumeAbrupt (*generator*, *abruptCompletion*, *generatorBrand*)

The abstract operation GeneratorResumeAbrupt takes arguments *generator*, *abruptCompletion* (a **Completion Record** whose [[Type]] is **return** or **throw**), and *generatorBrand*. It performs the following steps when called:

1. Let *state* be ? **GeneratorValidate**(*generator*, *generatorBrand*).
2. If *state* is **suspendedStart**, then
 - a. Set *generator*.[[GeneratorState]] to **completed**.
 - b. Once a generator enters the **completed** state it never leaves it and its associated **execution context** is never resumed. Any execution state associated with *generator* can be discarded at this point.
 - c. Set *state* to **completed**.
3. If *state* is **completed**, then
 - a. If *abruptCompletion*.[[Type]] is **return**, then
 - i. Return **CreateIterResultObject**(*abruptCompletion*.[[Value]], **true**).
 - b. Return **Completion**(*abruptCompletion*).
4. **Assert**: *state* is **suspendedYield**.
5. Let *genContext* be *generator*.[[GeneratorContext]].
6. Let *methodContext* be the **running execution context**.
7. Suspend *methodContext*.
8. Set *generator*.[[GeneratorState]] 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 *abruptCompletion* as the result of the operation that suspended it. Let *result* be the completion record 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 **Completion(result)**.

27.5.3.5 GetGeneratorKind ()

The abstract operation GetGeneratorKind takes no arguments. It performs the following steps when called:

1. Let *genContext* be the **running execution context**.
2. If *genContext* does not have a **Generator** component, return **non-generator**.
3. Let *generator* be the **Generator** component of *genContext*.
4. If *generator* has an [[AsyncGeneratorState]] internal slot, return **async**.
5. Else, return **sync**.

27.5.3.6 GeneratorYield (*iterNextObj*)

The abstract operation `GeneratorYield` takes argument *iterNextObj*. It performs the following steps when called:

1. **Assert:** *iterNextObj* is an Object that implements 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. **Assert:** `GetGeneratorKind()` is `sync`.
6. Set *generator*.[[GeneratorState]] to `suspendedYield`.
7. 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](#).
8. 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* that originally called this abstract operation.
9. Return `NormalCompletion(iterNextObj)`.
10. **NOTE:** This returns to the evaluation of the operation that had most previously resumed evaluation of *genContext*.

27.5.3.7 Yield (*value*)

The abstract operation `Yield` takes argument *value* (an [ECMAScript language value](#)). It performs the following steps when called:

1. Let *generatorKind* be `!GetGeneratorKind()`.
2. If *generatorKind* is `async`, return `?AsyncGeneratorYield(value)`.
3. Otherwise, return `?GeneratorYield(!CreateIterResultObject(value, false))`.

27.5.3.8 CreateIteratorFromClosure (*closure*, *generatorBrand*, *generatorPrototype*)

The abstract operation `CreateIteratorFromClosure` takes arguments *closure* (an [Abstract Closure](#) with no parameters), *generatorBrand*, and *generatorPrototype* (an Object). It performs the following steps when called:

1. **NOTE:** *closure* can contain uses of the [Yield](#) shorthand to yield an *IteratorResult* object.
2. Let *internalSlotsList* be « [[GeneratorState]], [[GeneratorContext]], [[GeneratorBrand]] ».
3. Let *generator* be `!OrdinaryObjectCreate(generatorPrototype, internalSlotsList)`.
4. Set *generator*.[[GeneratorBrand]] to *generatorBrand*.
5. Set *generator*.[[GeneratorState]] to `undefined`.
6. Perform `!GeneratorStart(generator, closure)`.
7. Return *generator*.

27.6 AsyncGenerator Objects

An `AsyncGenerator` object is an instance of an `async generator` function and conforms to both the `AsyncIterator` and `AsyncIterable` interfaces.

`AsyncGenerator` instances directly inherit properties from the object that is the initial value of the **"prototype"** property of the `AsyncGenerator` function that created the instance. `AsyncGenerator` instances indirectly inherit properties from the `AsyncGenerator` Prototype intrinsic, `%AsyncGeneratorFunction.prototype.prototype%`.

27.6.1 Properties of the AsyncGenerator Prototype Object

The *AsyncGenerator* prototype object:

- is *%AsyncGeneratorFunction.prototype.prototype%*.
- is an **ordinary object**.
- is not an AsyncGenerator instance and does not have an `[[AsyncGeneratorState]]` internal slot.
- has a `[[Prototype]]` internal slot whose value is *%AsyncIteratorPrototype%*.
- has properties that are indirectly inherited by all AsyncGenerator instances.

27.6.1.1 AsyncGenerator.prototype.constructor

The initial value of **AsyncGenerator.prototype.constructor** is *%AsyncGeneratorFunction.prototype%*.

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

27.6.1.2 AsyncGenerator.prototype.next (*value*)

1. Let *generator* be the **this** value.
2. Let *completion* be `NormalCompletion(value)`.
3. Return ! `AsyncGeneratorEnqueue(generator, completion, empty)`.

27.6.1.3 AsyncGenerator.prototype.return (*value*)

1. Let *generator* be the **this** value.
2. Let *completion* be `Completion` { `[[Type]]`: `return`, `[[Value]]`: *value*, `[[Target]]`: `empty` }.
3. Return ! `AsyncGeneratorEnqueue(generator, completion, empty)`.

27.6.1.4 AsyncGenerator.prototype.throw (*exception*)

1. Let *generator* be the **this** value.
2. Let *completion* be `ThrowCompletion(exception)`.
3. Return ! `AsyncGeneratorEnqueue(generator, completion, empty)`.

27.6.1.5 AsyncGenerator.prototype [@@toStringTag]

The initial value of the `@@toStringTag` property is the String value "AsyncGenerator".

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

27.6.2 Properties of AsyncGenerator Instances

AsyncGenerator instances are initially created with the internal slots described below:

Table 74: Internal Slots of AsyncGenerator Instances

Internal Slot	Description
[[AsyncGeneratorState]]	The current execution state of the async generator. The possible values are: undefined , suspendedStart , suspendedYield , executing , awaiting-return , and completed .
[[AsyncGeneratorContext]]	The execution context that is used when executing the code of this async generator.
[[AsyncGeneratorQueue]]	A List of AsyncGeneratorRequest records which represent requests to resume the async generator.
[[GeneratorBrand]]	A brand used to distinguish different kinds of async generators. The [[GeneratorBrand]] of async generators declared by ECMAScript source text is always empty.

27.6.3 AsyncGenerator Abstract Operations

27.6.3.1 AsyncGeneratorRequest Records

The AsyncGeneratorRequest is a **Record** value used to store information about how an async generator should be resumed and contains capabilities for fulfilling or rejecting the corresponding promise.

They have the following fields:

Table 75: AsyncGeneratorRequest Record Fields

Field Name	Value	Meaning
[[Completion]]	A Completion record	The completion which should be used to resume the async generator.
[[Capability]]	A PromiseCapability Record	The promise capabilities associated with this request.

27.6.3.2 AsyncGeneratorStart (*generator*, *generatorBody*)

The abstract operation AsyncGeneratorStart takes arguments *generator* and *generatorBody* (a **Parse Node** or an **Abstract Closure** with no parameters). It performs the following steps when called:

1. **Assert:** *generator* is an AsyncGenerator instance.
2. **Assert:** *generator*.[[AsyncGeneratorState]] is **undefined**.
3. Let *genContext* be the **running execution context**.
4. Set the Generator component of *genContext* to *generator*.
5. Set the code evaluation state of *genContext* such that when evaluation is resumed for that **execution context** the following steps will be performed:
 - a. If *generatorBody* is a **Parse Node**, then
 - i. Let *result* be the result of evaluating *generatorBody*.
 - b. Else,
 - i. **Assert:** *generatorBody* is an **Abstract Closure** with no parameters.

- ii. Let *result* be *generatorBody*(*generator*).
- c. **Assert**: If we return here, the async generator either threw an exception or performed either an implicit or explicit return.
- d. 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**.
- e. Set *generator*.[[AsyncGeneratorState]] to **completed**.
- f. If *result* is a normal completion, let *resultValue* be **undefined**.
- g. Else,
 - i. Let *resultValue* be *result*.[[Value]].
 - ii. If *result*.[[Type]] is not **return**, then
 - 1. Return ! **AsyncGeneratorReject**(*generator*, *resultValue*).
- h. Return ! **AsyncGeneratorResolve**(*generator*, *resultValue*, **true**).
- 6. Set *generator*.[[AsyncGeneratorContext]] to *genContext*.
- 7. Set *generator*.[[AsyncGeneratorState]] to **suspendedStart**.
- 8. Set *generator*.[[AsyncGeneratorQueue]] to a new empty **List**.
- 9. Return **undefined**.

27.6.3.3 AsyncGeneratorValidate (*generator*, *generatorBrand*)

The abstract operation AsyncGeneratorValidate takes arguments *generator* and *generatorBrand*. It performs the following steps when called:

1. Perform ? **RequireInternalSlot**(*generator*, [[AsyncGeneratorContext]]).
2. Perform ? **RequireInternalSlot**(*generator*, [[AsyncGeneratorState]]).
3. Perform ? **RequireInternalSlot**(*generator*, [[AsyncGeneratorQueue]]).
4. If *generator*.[[GeneratorBrand]] is not the same value as *generatorBrand*, throw a **TypeError** exception.

27.6.3.4 AsyncGeneratorResolve (*generator*, *value*, *done*)

The abstract operation AsyncGeneratorResolve takes arguments *generator*, *value*, and *done* (a Boolean). It performs the following steps when called:

1. **Assert**: *generator* is an AsyncGenerator instance.
2. Let *queue* be *generator*.[[AsyncGeneratorQueue]].
3. **Assert**: *queue* is not an empty **List**.
4. Let *next* be the first element of *queue*.
5. Remove the first element from *queue*.
6. Let *promiseCapability* be *next*.[[Capability]].
7. Let *iteratorResult* be ! **CreateIterResultObject**(*value*, *done*).
8. Perform ! **Call**(*promiseCapability*.[[Resolve]], **undefined**, « *iteratorResult* »).
9. Perform ! **AsyncGeneratorResumeNext**(*generator*).
10. Return **undefined**.

27.6.3.5 AsyncGeneratorReject (*generator*, *exception*)

The abstract operation AsyncGeneratorReject takes arguments *generator* and *exception*. It performs the following steps when called:

1. **Assert**: *generator* is an AsyncGenerator instance.

2. Let *queue* be *generator*.[[AsyncGeneratorQueue]].
3. **Assert:** *queue* is not an empty **List**.
4. Let *next* be the first element of *queue*.
5. Remove the first element from *queue*.
6. Let *promiseCapability* be *next*.[[Capability]].
7. Perform ! **Call**(*promiseCapability*.[[Reject]], **undefined**, « *exception* »).
8. Perform ! **AsyncGeneratorResumeNext**(*generator*).
9. Return **undefined**.

27.6.3.6 AsyncGeneratorResumeNext (*generator*)

The abstract operation AsyncGeneratorResumeNext takes argument *generator*. It performs the following steps when called:

1. **Assert:** *generator* is an AsyncGenerator instance.
2. Let *state* be *generator*.[[AsyncGeneratorState]].
3. **Assert:** *state* is not **executing**.
4. If *state* is **awaiting-return**, return **undefined**.
5. Let *queue* be *generator*.[[AsyncGeneratorQueue]].
6. If *queue* is an empty **List**, return **undefined**.
7. Let *next* be the value of the first element of *queue*.
8. **Assert:** *next* is an AsyncGeneratorRequest record.
9. Let *completion* be *next*.[[Completion]].
10. If *completion* is an **abrupt completion**, then
 - a. If *state* is **suspendedStart**, then
 - i. Set *generator*.[[AsyncGeneratorState]] to **completed**.
 - ii. Set *state* to **completed**.
 - b. If *state* is **completed**, then
 - i. If *completion*.[[Type]] is **return**, then
 1. Set *generator*.[[AsyncGeneratorState]] to **awaiting-return**.
 2. Let *promise* be ? **PromiseResolve**(%*Promise*%, *completion*.[[Value]]).
 3. Let *stepsFulfilled* be the algorithm steps defined in **AsyncGeneratorResumeNext Return Processor Fulfilled Functions**.
 4. Let *lengthFulfilled* be the number of non-optional parameters of the function definition in **AsyncGeneratorResumeNext Return Processor Fulfilled Functions**.
 5. Let *onFulfilled* be ! **CreateBuiltinFunction**(*stepsFulfilled*, *lengthFulfilled*, "", « [[Generator]] »).
 6. Set *onFulfilled*.[[Generator]] to *generator*.
 7. Let *stepsRejected* be the algorithm steps defined in **AsyncGeneratorResumeNext Return Processor Rejected Functions**.
 8. Let *lengthRejected* be the number of non-optional parameters of the function definition in **AsyncGeneratorResumeNext Return Processor Rejected Functions**.
 9. Let *onRejected* be ! **CreateBuiltinFunction**(*stepsRejected*, *lengthRejected*, "", « [[Generator]] »).
 10. Set *onRejected*.[[Generator]] to *generator*.
 11. Perform ! **PerformPromiseThen**(*promise*, *onFulfilled*, *onRejected*).
 12. Return **undefined**.
 - ii. Else,
 1. **Assert:** *completion*.[[Type]] is **throw**.
 2. Perform ! **AsyncGeneratorReject**(*generator*, *completion*.[[Value]]).

3. Return **undefined**.

11. Else if *state* is completed, return ! `AsyncGeneratorResolve(generator, undefined, true)`.
12. **Assert:** *state* is either `suspendedStart` or `suspendedYield`.
13. Let *genContext* be *generator*.`[[AsyncGeneratorContext]]`.
14. Let *callerContext* be the `running execution context`.
15. Suspend *callerContext*.
16. Set *generator*.`[[AsyncGeneratorState]]` to `executing`.
17. Push *genContext* onto the `execution context stack`; *genContext* is now the `running execution context`.
18. Resume the suspended evaluation of *genContext* using *completion* as the result of the operation that suspended it. Let *result* be the completion record returned by the resumed computation.
19. **Assert:** *result* is never an `abrupt completion`.
20. **Assert:** When we return here, *genContext* has already been removed from the `execution context stack` and *callerContext* is the currently `running execution context`.
21. Return **undefined**.

27.6.3.6.1 AsyncGeneratorResumeNext Return Processor Fulfilled Functions

An `AsyncGeneratorResumeNext` return processor fulfilled function is an anonymous built-in function that is used as part of the `AsyncGeneratorResumeNext` specification device to unwrap promises passed in to the `AsyncGenerator.prototype.return (value)` method. Each `AsyncGeneratorResumeNext` return processor fulfilled function has a `[[Generator]]` internal slot.

When an `AsyncGeneratorResumeNext` return processor fulfilled function is called with argument *value*, the following steps are taken:

1. Let *F* be the `active function object`.
2. Set *F*.`[[Generator]]`.`[[AsyncGeneratorState]]` to `completed`.
3. Return ! `AsyncGeneratorResolve(F. [[Generator]], value, true)`.

The `"length"` property of an `AsyncGeneratorResumeNext` return processor fulfilled function is `1`.

27.6.3.6.2 AsyncGeneratorResumeNext Return Processor Rejected Functions

An `AsyncGeneratorResumeNext` return processor rejected function is an anonymous built-in function that is used as part of the `AsyncGeneratorResumeNext` specification device to unwrap promises passed in to the `AsyncGenerator.prototype.return (value)` method. Each `AsyncGeneratorResumeNext` return processor rejected function has a `[[Generator]]` internal slot.

When an `AsyncGeneratorResumeNext` return processor rejected function is called with argument *reason*, the following steps are taken:

1. Let *F* be the `active function object`.
2. Set *F*.`[[Generator]]`.`[[AsyncGeneratorState]]` to `completed`.
3. Return ! `AsyncGeneratorReject(F. [[Generator]], reason)`.

The `"length"` property of an `AsyncGeneratorResumeNext` return processor rejected function is `1`.

27.6.3.7 AsyncGeneratorEnqueue (generator, completion, generatorBrand)

The abstract operation `AsyncGeneratorEnqueue` takes arguments *generator*, *completion* (a `Completion Record`), and

generatorBrand. It performs the following steps when called:

1. Let *promiseCapability* be ! **NewPromiseCapability**(%Promise%).
2. Let *check* be **AsyncGeneratorValidate**(*generator*, *generatorBrand*).
3. If *check* is an **abrupt completion**, then
 - a. Let *badGeneratorError* be a newly created **TypeError** object.
 - b. Perform ! **Call**(*promiseCapability*.[[Reject]], **undefined**, « *badGeneratorError* »).
 - c. Return *promiseCapability*.[[Promise]].
4. Let *queue* be *generator*.[[AsyncGeneratorQueue]].
5. Let *request* be **AsyncGeneratorRequest** { [[Completion]]: *completion*, [[Capability]]: *promiseCapability* }.
6. Append *request* to the end of *queue*.
7. Let *state* be *generator*.[[AsyncGeneratorState]].
8. If *state* is not **executing**, then
 - a. Perform ! **AsyncGeneratorResumeNext**(*generator*).
9. Return *promiseCapability*.[[Promise]].

27.6.3.8 AsyncGeneratorYield (*value*)

The abstract operation AsyncGeneratorYield takes argument *value*. It performs the following steps when called:

1. Let *genContext* be the **running execution context**.
2. **Assert**: *genContext* is the **execution context** of a generator.
3. Let *generator* be the value of the Generator component of *genContext*.
4. **Assert**: **GetGeneratorKind**() is **async**.
5. Set *value* to ? **Await**(*value*).
6. Set *generator*.[[AsyncGeneratorState]] to **suspendedYield**.
7. 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**.
8. Set the code evaluation state of *genContext* such that when evaluation is resumed with a **Completion resumptionValue** the following steps will be performed:
 - a. If *resumptionValue*.[[Type]] is not **return**, return **Completion**(*resumptionValue*).
 - b. Let *awaited* be **Await**(*resumptionValue*.[[Value]]).
 - c. If *awaited*.[[Type]] is **throw**, return **Completion**(*awaited*).
 - d. **Assert**: *awaited*.[[Type]] is **normal**.
 - e. Return **Completion** { [[Type]]: **return**, [[Value]]: *awaited*.[[Value]], [[Target]]: **empty** }.
 - f. NOTE: When one of the above steps returns, it returns to the evaluation of the *YieldExpression* production that originally called this abstract operation.
9. Return ! **AsyncGeneratorResolve**(*generator*, *value*, **false**).
10. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of *genContext*.

27.6.3.9 CreateAsyncIteratorFromClosure (*closure*, *generatorBrand*, *generatorPrototype*)

The abstract operation CreateAsyncIteratorFromClosure takes arguments *closure* (an **Abstract Closure** with no parameters), *generatorBrand*, and *generatorPrototype* (an Object). It performs the following steps when called:

1. NOTE: *closure* can contain uses of the **Await** shorthand and uses of the **Yield** shorthand to yield an **IteratorResult** object.
2. Let *internalSlotsList* be « [[AsyncGeneratorState]], [[AsyncGeneratorContext]], [[AsyncGeneratorQueue]],

- [[GeneratorBrand]] ».
- 3. Let *generator* be ! OrdinaryObjectCreate(*generatorPrototype*, *internalSlotsList*).
- 4. Set *generator*.[[GeneratorBrand]] to *generatorBrand*.
- 5. Set *generator*.[[AsyncGeneratorState]] to **undefined**.
- 6. Perform ! AsyncGeneratorStart(*generator*, *closure*).
- 7. Return *generator*.

27.7 AsyncFunction Objects

AsyncFunction objects are functions that are usually created by evaluating *AsyncFunctionDeclarations*, *AsyncFunctionExpressions*, *AsyncMethods*, and *AsyncArrowFunctions*. They may also be created by calling the `%AsyncFunction%` intrinsic.

27.7.1 The AsyncFunction Constructor

The AsyncFunction `constructor`:

- is `%AsyncFunction%`.
- is a subclass of **Function**.
- creates and initializes a new AsyncFunction object when called as a function rather than as a `constructor`. Thus the function call **AsyncFunction(…)** is equivalent to the object creation expression **new AsyncFunction(…)** with the same arguments.
- is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified AsyncFunction behaviour must include a **super** call to the AsyncFunction `constructor` to create and initialize a subclass instance with the internal slots necessary for built-in async function behaviour. All ECMAScript syntactic forms for defining async function objects create direct instances of AsyncFunction. There is no syntactic means to create instances of AsyncFunction subclasses.

27.7.1.1 AsyncFunction (*p1*, *p2*, … , *pn*, *body*)

The last argument specifies the body (executable code) of an async function. Any preceding arguments specify formal parameters.

When the **AsyncFunction** 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 *C* be the `active function object`.
2. Let *args* be the `argumentsList` that was passed to this function by `[[Call]]` or `[[Construct]]`.
3. Return `CreateDynamicFunction(C, NewTarget, async, args)`.

NOTE See NOTE for 20.2.1.1.

27.7.2 Properties of the AsyncFunction Constructor

The AsyncFunction `constructor`:

- is a standard built-in `function object` that inherits from the Function `constructor`.

- has a `[[Prototype]]` internal slot whose value is `%Function%`.
- has a `"name"` property whose value is `"AsyncFunction"`.
- has the following properties:

27.7.2.1 `AsyncFunction.length`

This is a [data property](#) with a value of 1. This property has the attributes { `[[Writable]]: false`, `[[Enumerable]]: false`, `[[Configurable]]: true` }.

27.7.2.2 `AsyncFunction.prototype`

The initial value of `AsyncFunction.prototype` is the [AsyncFunction prototype object](#).

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

27.7.3 Properties of the AsyncFunction Prototype Object

The *AsyncFunction prototype object*:

- is `%AsyncFunction.prototype%`.
- is an [ordinary object](#).
- is not a [function object](#) and does not have an `[[ECMAScriptCode]]` internal slot or any other of the internal slots listed in [Table 29](#).
- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.

27.7.3.1 `AsyncFunction.prototype.constructor`

The initial value of `AsyncFunction.prototype.constructor` is `%AsyncFunction%`

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

27.7.3.2 `AsyncFunction.prototype [@@toStringTag]`

The initial value of the `@@toStringTag` property is the String value `"AsyncFunction"`.

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

27.7.4 AsyncFunction Instances

Every AsyncFunction instance is an ECMAScript [function object](#) and has the internal slots listed in [Table 29](#). The value of the `[[IsClassConstructor]]` internal slot for all such instances is `false`. AsyncFunction instances are not constructors and do not have a `[[Construct]]` internal method. AsyncFunction instances do not have a prototype property as they are not constructible.

Each AsyncFunction instance has the following own properties:

27.7.4.1 `length`

The specification for the `"length"` property of Function instances given in [20.2.4.1](#) also applies to AsyncFunction

instances.

27.7.4.2 name

The specification for the "name" property of Function instances given in 20.2.4.2 also applies to AsyncFunction instances.

27.7.5 Async Functions Abstract Operations

27.7.5.1 AsyncFunctionStart (*promiseCapability*, *asyncFunctionBody*)

The abstract operation AsyncFunctionStart takes arguments *promiseCapability* (a [PromiseCapability Record](#)) and *asyncFunctionBody*. It performs the following steps when called:

1. Let *runningContext* be the [running execution context](#).
2. Let *asyncContext* be a copy of *runningContext*.
3. NOTE: Copying the execution state is required for the step below to resume its execution. It is ill-defined to resume a currently executing context.
4. Set the code evaluation state of *asyncContext* such that when evaluation is resumed for that [execution context](#) the following steps will be performed:
 - a. Let *result* be the result of evaluating *asyncFunctionBody*.
 - b. **Assert**: If we return here, the async function either threw an exception or performed an implicit or explicit return; all awaiting is done.
 - c. Remove *asyncContext* 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. If *result*.[[Type]] is normal, then
 - i. Perform ! [Call](#)(*promiseCapability*.[[Resolve]], **undefined**, « **undefined** »).
 - e. Else if *result*.[[Type]] is return, then
 - i. Perform ! [Call](#)(*promiseCapability*.[[Resolve]], **undefined**, « *result*.[[Value]] »).
 - f. Else,
 - i. **Assert**: *result*.[[Type]] is throw.
 - ii. Perform ! [Call](#)(*promiseCapability*.[[Reject]], **undefined**, « *result*.[[Value]] »).
 - g. Return.
5. Push *asyncContext* onto the [execution context stack](#); *asyncContext* is now the [running execution context](#).
6. Resume the suspended evaluation of *asyncContext*. Let *result* be the value returned by the resumed computation.
7. **Assert**: When we return here, *asyncContext* has already been removed from the [execution context stack](#) and *runningContext* is the currently [running execution context](#).
8. **Assert**: *result* is a normal completion with a value of **undefined**. The possible sources of completion values are [Await](#) or, if the async function doesn't await anything, step 4.g above.
9. Return.

28 Reflection

28.1 The Reflect Object

The Reflect object:

- is `%Reflect%`.
- is the initial value of the "Reflect" property of the [global object](#).
- is an [ordinary object](#).
- has a `[[Prototype]]` internal slot whose value is `%Object.prototype%`.
- is not a [function object](#).
- does not have a `[[Construct]]` internal method; it cannot be used as a [constructor](#) with the **new** operator.
- does not have a `[[Call]]` internal method; it cannot be invoked as a function.

28.1.1 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**, throw a **TypeError** exception.
2. Let *args* be `? CreateListFromArrayLike(argumentsList)`.
3. Perform `PrepareForTailCall()`.
4. Return `? Call(target, thisArgument, args)`.

28.1.2 Reflect.construct (*target*, *argumentsList* [, *newTarget*])

When the **construct** function is called with arguments *target*, *argumentsList*, and *newTarget*, the following steps are taken:

1. If `IsConstructor(target)` is **false**, throw a **TypeError** exception.
2. If *newTarget* is not present, set *newTarget* to *target*.
3. Else if `IsConstructor(newTarget)` is **false**, throw a **TypeError** exception.
4. Let *args* be `? CreateListFromArrayLike(argumentsList)`.
5. Return `? Construct(target, args, newTarget)`.

28.1.3 Reflect.defineProperty (*target*, *propertyKey*, *attributes*)

When the **defineProperty** function is called with arguments *target*, *propertyKey*, and *attributes*, the following steps are taken:

1. If `Type(target)` is not Object, throw a **TypeError** exception.
2. Let *key* be `? ToPropertyKey(propertyKey)`.
3. Let *desc* be `? ToPropertyDescriptor(attributes)`.
4. Return `? target.[[DefineOwnProperty]](key, desc)`.

28.1.4 Reflect.deleteProperty (*target*, *propertyKey*)

When the **deleteProperty** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. If `Type(target)` is not Object, throw a **TypeError** exception.

2. Let *key* be ? `ToPropertyKey`(*propertyKey*).
3. Return ? *target*.`[[Delete]]`(*key*).

28.1.5 `Reflect.get` (*target*, *propertyKey* [, *receiver*])

When the **get** function is called with arguments *target*, *propertyKey*, and *receiver*, the following steps are taken:

1. If `Type`(*target*) is not Object, throw a **TypeError** exception.
2. Let *key* be ? `ToPropertyKey`(*propertyKey*).
3. If *receiver* is not present, then
 - a. Set *receiver* to *target*.
4. Return ? *target*.`[[Get]]`(*key*, *receiver*).

28.1.6 `Reflect.getOwnPropertyDescriptor` (*target*, *propertyKey*)

When the **getOwnPropertyDescriptor** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. If `Type`(*target*) is not Object, throw a **TypeError** exception.
2. Let *key* be ? `ToPropertyKey`(*propertyKey*).
3. Let *desc* be ? *target*.`[[GetOwnProperty]]`(*key*).
4. Return `FromPropertyDescriptor`(*desc*).

28.1.7 `Reflect.getPrototypeOf` (*target*)

When the **getPrototypeOf** function is called with argument *target*, the following steps are taken:

1. If `Type`(*target*) is not Object, throw a **TypeError** exception.
2. Return ? *target*.`[[GetPrototypeOf]]`()().

28.1.8 `Reflect.has` (*target*, *propertyKey*)

When the **has** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. If `Type`(*target*) is not Object, throw a **TypeError** exception.
2. Let *key* be ? `ToPropertyKey`(*propertyKey*).
3. Return ? *target*.`[[HasProperty]]`(*key*).

28.1.9 `Reflect.isExtensible` (*target*)

When the **isExtensible** function is called with argument *target*, the following steps are taken:

1. If `Type`(*target*) is not Object, throw a **TypeError** exception.
2. Return ? *target*.`[[IsExtensible]]`()().

28.1.10 `Reflect.ownKeys` (*target*)

When the **ownKeys** function is called with argument *target*, the following steps are taken:

1. If `Type(target)` is not Object, throw a **TypeError** exception.
2. Let *keys* be ? `target.[[OwnPropertyKeys]]()`.
3. Return `CreateArrayFromList(keys)`.

28.1.11 Reflect.preventExtensions (*target*)

When the **preventExtensions** function is called with argument *target*, the following steps are taken:

1. If `Type(target)` is not Object, throw a **TypeError** exception.
2. Return ? `target.[[PreventExtensions]]()`.

28.1.12 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. If `Type(target)` is not Object, throw a **TypeError** exception.
2. Let *key* be ? `ToPropertyKey(propertyKey)`.
3. If *receiver* is not present, then
 - a. Set *receiver* to *target*.
4. Return ? `target.[[Set]](key, V, receiver)`.

28.1.13 Reflect.setPrototypeOf (*target*, *proto*)

When the **setPrototypeOf** function is called with arguments *target* and *proto*, the following steps are taken:

1. If `Type(target)` is not Object, throw a **TypeError** exception.
2. If `Type(proto)` is not Object and *proto* is not **null**, throw a **TypeError** exception.
3. Return ? `target.[[SetPrototypeOf]](proto)`.

28.1.14 Reflect [@@toStringTag]

The initial value of the `@@toStringTag` property is the String value **"Reflect"**.

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

28.2 Proxy Objects

28.2.1 The Proxy Constructor

The Proxy **constructor**:

- is `%Proxy%`.
- is the initial value of the **"Proxy"** property of the **global object**.
- creates and initializes a new **Proxy exotic object** when called as a **constructor**.
- is not intended to be called as a function and will throw an exception when called in that manner.

28.2.1.1 Proxy (*target*, *handler*)

When **Proxy** is called with arguments *target* and *handler*, it performs the following steps:

1. If *NewTarget* is **undefined**, throw a **TypeError** exception.
2. Return ? **ProxyCreate**(*target*, *handler*).

28.2.2 Properties of the Proxy Constructor

The Proxy **constructor**:

- has a `[[Prototype]]` internal slot whose value is `%Function.prototype%`.
- does not have a **"prototype"** property because Proxy exotic objects do not have a `[[Prototype]]` internal slot that requires initialization.
- has the following properties:

28.2.2.1 Proxy.revocable (*target*, *handler*)

The **Proxy.revocable** function is used to create a revocable Proxy object. When **Proxy.revocable** is called with arguments *target* and *handler*, the following steps are taken:

1. Let *p* be ? **ProxyCreate**(*target*, *handler*).
2. Let *steps* be the algorithm steps defined in **Proxy Revocation Functions**.
3. Let *length* be the number of non-optional parameters of the function definition in **Proxy Revocation Functions**.
4. Let *revoker* be ! **CreateBuiltinFunction**(*steps*, *length*, "", « `[[RevocableProxy]]` »).
5. Set *revoker*.`[[RevocableProxy]]` to *p*.
6. Let *result* be ! **OrdinaryObjectCreate**(`%Object.prototype%`).
7. Perform ! **CreateDataPropertyOrThrow**(*result*, **"proxy"**, *p*).
8. Perform ! **CreateDataPropertyOrThrow**(*result*, **"revoke"**, *revoker*).
9. Return *result*.

28.2.2.1.1 Proxy Revocation Functions

A Proxy revocation function is an anonymous built-in function that has the ability to invalidate a specific Proxy object.

Each Proxy revocation function has a `[[RevocableProxy]]` internal slot.

When a Proxy revocation function is called, the following steps are taken:

1. Let *F* be the **active function object**.
2. Let *p* be *F*.`[[RevocableProxy]]`.
3. If *p* is **null**, return **undefined**.
4. Set *F*.`[[RevocableProxy]]` to **null**.
5. **Assert**: *p* is a Proxy object.
6. Set *p*.`[[ProxyTarget]]` to **null**.
7. Set *p*.`[[ProxyHandler]]` to **null**.
8. Return **undefined**.

The **"length"** property of a Proxy revocation function is **+0_F**.

28.3 Module Namespace Objects

A Module Namespace Object is a [module namespace exotic object](#) that provides runtime property-based access to a module's exported bindings. There is no [constructor](#) function for Module Namespace Objects. Instead, such an object is created for each module that is imported by an *ImportDeclaration* that includes a *NamespaceImport*.

In addition to the properties specified in [10.4.6](#) each Module Namespace Object has the following own property:

28.3.1 @@toStringTag

The initial value of the [@@toStringTag](#) property is the String value "Module".

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

29 Memory Model

The memory consistency model, or *memory model*, specifies the possible orderings of [Shared Data Block](#) events, arising via accessing TypedArray instances backed by a SharedArrayBuffer and via methods on the Atomics object. When the program has no data races (defined below), the ordering of events appears as sequentially consistent, i.e., as an interleaving of actions from each [agent](#). When the program has data races, shared memory operations may appear sequentially inconsistent. For example, programs may exhibit causality-violating behaviour and other astonishments. These astonishments arise from compiler transforms and the design of CPUs (e.g., out-of-order execution and speculation). The memory model defines both the precise conditions under which a program exhibits sequentially consistent behaviour as well as the possible values read from data races. To wit, there is no undefined behaviour.

The memory model is defined as relational constraints on events introduced by [abstract operations](#) on SharedArrayBuffer or by methods on the Atomics object during an evaluation.

NOTE

This section provides an axiomatic model on events introduced by the [abstract operations](#) on SharedArrayBuffers. It bears stressing that the model is not expressible algorithmically, unlike the rest of this specification. The nondeterministic introduction of events by [abstract operations](#) is the interface between the operational semantics of ECMAScript evaluation and the axiomatic semantics of the memory model. The semantics of these events is defined by considering graphs of all events in an evaluation. These are neither Static Semantics nor Runtime Semantics. There is no demonstrated algorithmic implementation, but instead a set of constraints that determine if a particular event graph is allowed or disallowed.

29.1 Memory Model Fundamentals

Shared memory accesses (reads and writes) are divided into two groups, atomic accesses and data accesses, defined below. Atomic accesses are sequentially consistent, i.e., there is a strict total ordering of events agreed upon by all agents in an [agent cluster](#). Non-atomic accesses do not have a strict total ordering agreed upon by all agents, i.e., unordered.

NOTE 1

No orderings weaker than sequentially consistent and stronger than unordered, such as release-acquire, are supported.

A *Shared Data Block* event is either a *ReadSharedMemory*, *WriteSharedMemory*, or *ReadModifyWriteSharedMemory* **Record**.

Table 76: ReadSharedMemory Event Fields

Field Name	Value	Meaning
[[Order]]	SeqCst Unordered	The weakest ordering guaranteed by the memory model for the event.
[[NoTear]]	A Boolean	Whether this event is allowed to read from multiple write events on equal range as this event.
[[Block]]	A Shared Data Block	The block the event operates on.
[[ByteIndex]]	A non-negative integer	The byte address of the read in [[Block]].
[[ElementSize]]	A non-negative integer	The size of the read.

Table 77: WriteSharedMemory Event Fields

Field Name	Value	Meaning
[[Order]]	SeqCst Unordered Init	The weakest ordering guaranteed by the memory model for the event.
[[NoTear]]	A Boolean	Whether this event is allowed to be read from multiple read events with equal range as this event.
[[Block]]	A Shared Data Block	The block the event operates on.
[[ByteIndex]]	A non-negative integer	The byte address of the write in [[Block]].
[[ElementSize]]	A non-negative integer	The size of the write.
[[Payload]]	A List	The List of byte values to be read by other events.

Table 78: ReadModifyWriteSharedMemory Event Fields

Field Name	Value	Meaning
[[Order]]	SeqCst	Read-modify-write events are always sequentially consistent.
[[NoTear]]	true	Read-modify-write events cannot tear.
[[Block]]	A Shared Data Block	The block the event operates on.
[[ByteIndex]]	A non-negative integer	The byte address of the read-modify-write in [[Block]].
[[ElementSize]]	A non-negative integer	The size of the read-modify-write.
[[Payload]]	A List	The List of byte values to be passed to [[ModifyOp]].
[[ModifyOp]]	A read-modify-write modification function	An abstract closure that returns a modified List of byte values from a read List of byte values and [[Payload]].

These events are introduced by [abstract operations](#) or by methods on the `Atomics` object.

Some operations may also introduce *Synchronize* events. A *Synchronize event* has no fields, and exists purely to directly constrain the permitted orderings of other events.

In addition to [Shared Data Block](#) and *Synchronize* events, there are [host-specific](#) events.

Let the range of a `ReadSharedMemory`, `WriteSharedMemory`, or `ReadModifyWriteSharedMemory` event be the Set of contiguous integers from its `[[ByteIndex]]` to `[[ByteIndex]] + [[ElementSize]] - 1`. Two events' ranges are equal when the events have the same `[[Block]]`, and the ranges are element-wise equal. Two events' ranges are overlapping when the events have the same `[[Block]]`, the ranges are not equal and their intersection is non-empty. Two events' ranges are disjoint when the events do not have the same `[[Block]]` or their ranges are neither equal nor overlapping.

NOTE 2

Examples of [host-specific](#) synchronizing events that should be accounted for are: sending a `SharedArrayBuffer` from one [agent](#) to another (e.g., by `postMessage` in a browser), starting and stopping agents, and communicating within the [agent cluster](#) via channels other than shared memory. It is assumed those events are appended to [agent-order](#) during evaluation like the other `SharedArrayBuffer` events.

Events are ordered within candidate executions by the relations defined below.

29.2 Agent Events Records

An *Agent Events Record* is a [Record](#) with the following fields.

Table 79: Agent Events Record Fields

Field Name	Value	Meaning
[[AgentSignifier]]	A value that admits equality testing	The agent whose evaluation resulted in this ordering.
[[EventList]]	A List of events	Events are appended to the list during evaluation.
[[AgentSynchronizesWith]]	A List of pairs of Synchronize events	Synchronize relationships introduced by the operational semantics.

29.3 Chosen Value Records

A *Chosen Value Record* is a [Record](#) with the following fields.

Table 80: Chosen Value Record Fields

Field Name	Value	Meaning
[[Event]]	A Shared Data Block event	The ReadSharedMemory or ReadModifyWriteSharedMemory event that was introduced for this chosen value.
[[ChosenValue]]	A List of byte values	The bytes that were nondeterministically chosen during evaluation.

29.4 Candidate Executions

A *candidate execution* of the evaluation of an [agent cluster](#) is a [Record](#) with the following fields.

Table 81: Candidate Execution Record Fields

Field Name	Value	Meaning
[[EventsRecords]]	A List of Agent Events Records.	Maps an agent to Lists of events appended during the evaluation.
[[ChosenValues]]	A List of Chosen Value Records.	Maps ReadSharedMemory or ReadModifyWriteSharedMemory events to the List of byte values chosen during the evaluation.
[[AgentOrder]]	An agent-order Relation.	Defined below.
[[ReadsBytesFrom]]	A reads-bytes-from mathematical function.	Defined below.
[[ReadsFrom]]	A reads-from Relation.	Defined below.
[[HostSynchronizesWith]]	A host-synchronizes-with Relation.	Defined below.
[[SynchronizesWith]]	A synchronizes-with Relation.	Defined below.
[[HappensBefore]]	A happens-before Relation.	Defined below.

An *empty candidate execution* is a candidate execution Record whose fields are empty Lists and Relations.

29.5 Abstract Operations for the Memory Model

29.5.1 EventSet (*execution*)

The abstract operation EventSet takes argument *execution* (a candidate execution). It performs the following steps when called:

1. Let *events* be an empty Set.
2. For each Agent Events Record *aer* of *execution*.[[EventsRecords]], do
 - a. For each event *E* of *aer*.[[EventList]], do
 - i. Add *E* to *events*.
3. Return *events*.

29.5.2 SharedDataBlockEventSet (*execution*)

The abstract operation SharedDataBlockEventSet takes argument *execution* (a candidate execution). It performs the following steps when called:

1. Let *events* be an empty Set.

2. For each event E of `EventSet(execution)`, do
 - a. If E is a `ReadSharedMemory`, `WriteSharedMemory`, or `ReadModifyWriteSharedMemory` event, add E to $events$.
3. Return $events$.

29.5.3 HostEventSet (*execution*)

The abstract operation `HostEventSet` takes argument $execution$ (a `candidate execution`). It performs the following steps when called:

1. Let $events$ be an empty Set.
2. For each event E of `EventSet(execution)`, do
 - a. If E is not in `SharedDataBlockEventSet(execution)`, add E to $events$.
3. Return $events$.

29.5.4 ComposeWriteEventBytes (*execution, byteIndex, Ws*)

The abstract operation `ComposeWriteEventBytes` takes arguments $execution$ (a `candidate execution`), $byteIndex$ (a non-negative `integer`), and Ws (a `List` of `WriteSharedMemory` or `ReadModifyWriteSharedMemory` events). It performs the following steps when called:

1. Let $byteLocation$ be $byteIndex$.
2. Let $bytesRead$ be a new empty `List`.
3. For each element W of Ws , do
 - a. **Assert:** W has $byteLocation$ in its range.
 - b. Let $payloadIndex$ be $byteLocation - W.[[ByteIndex]]$.
 - c. If W is a `WriteSharedMemory` event, then
 - i. Let $byte$ be $W.[[Payload]][[payloadIndex]]$.
 - d. Else,
 - i. **Assert:** W is a `ReadModifyWriteSharedMemory` event.
 - ii. Let $bytes$ be `ValueOfReadEvent(execution, W)`.
 - iii. Let $bytesModified$ be $W.[[ModifyOp]](bytes, W.[[Payload]])$.
 - iv. Let $byte$ be $bytesModified[payloadIndex]$.
 - e. Append $byte$ to $bytesRead$.
 - f. Set $byteLocation$ to $byteLocation + 1$.
4. Return $bytesRead$.

NOTE 1 The read-modify-write modification `[[ModifyOp]]` is given by the function properties on the `Atomics` object that introduce `ReadModifyWriteSharedMemory` events.

NOTE 2 This abstract operation composes a `List` of write events into a `List` of byte values. It is used in the event semantics of `ReadSharedMemory` and `ReadModifyWriteSharedMemory` events.

29.5.5 ValueOfReadEvent (*execution, R*)

The abstract operation `ValueOfReadEvent` takes arguments $execution$ (a `candidate execution`) and R (a

ReadSharedMemory or ReadModifyWriteSharedMemory event). It performs the following steps when called:

1. Assert: R is a ReadSharedMemory or ReadModifyWriteSharedMemory event.
2. Let Ws be $execution.[[ReadsBytesFrom]](R)$.
3. Assert: Ws is a List of WriteSharedMemory or ReadModifyWriteSharedMemory events with length equal to $R.[[ElementSize]]$.
4. Return $ComposeWriteEventBytes(execution, R.[[ByteIndex]], Ws)$.

29.6 Relations of Candidate Executions

29.6.1 agent-order

For a candidate execution $execution$, $execution.[[AgentOrder]]$ is a Relation on events that satisfies the following.

- For each pair (E, D) in $EventSet(execution)$, (E, D) is in $execution.[[AgentOrder]]$ if there is some Agent Events Record aer in $execution.[[EventsRecords]]$ such that E and D are in $aer.[[EventList]]$ and E is before D in List order of $aer.[[EventList]]$.

NOTE

Each agent introduces events in a per-agent strict total order during the evaluation. This is the union of those strict total orders.

29.6.2 reads-bytes-from

For a candidate execution $execution$, $execution.[[ReadsBytesFrom]]$ is a mathematical function mapping events in $SharedDataBlockEventSet(execution)$ to Lists of events in $SharedDataBlockEventSet(execution)$ that satisfies the following conditions.

- For each ReadSharedMemory or ReadModifyWriteSharedMemory event R in $SharedDataBlockEventSet(execution)$, $execution.[[ReadsBytesFrom]](R)$ is a List of length $R.[[ElementSize]]$ whose elements are WriteSharedMemory or ReadModifyWriteSharedMemory events Ws such that all of the following are true.
 - Each event W with index i in Ws has $R.[[ByteIndex]] + i$ in its range.
 - R is not in Ws .

29.6.3 reads-from

For a candidate execution $execution$, $execution.[[ReadsFrom]]$ is the least Relation on events that satisfies the following.

- For each pair (R, W) in $SharedDataBlockEventSet(execution)$, (R, W) is in $execution.[[ReadsFrom]]$ if W is in $execution.[[ReadsBytesFrom]](R)$.

29.6.4 host-synchronizes-with

For a candidate execution $execution$, $execution.[[HostSynchronizesWith]]$ is a host-provided strict partial order on host-specific events that satisfies at least the following.

- If (E, D) is in $execution.[[HostSynchronizesWith]]$, E and D are in $HostEventSet(execution)$.
- There is no cycle in the union of $execution.[[HostSynchronizesWith]]$ and $execution.[[AgentOrder]]$.

NOTE 1 For two **host-specific** events E and D , E **host-synchronizes-with** D implies E **happens-before** D .

NOTE 2 The **host-synchronizes-with** relation allows the **host** to provide additional synchronization mechanisms, such as **postMessage** between HTML workers.

29.6.5 synchronizes-with

For a **candidate execution** $execution$, $execution.[[SynchronizesWith]]$ is the least **Relation** on events that satisfies the following.

- For each pair (R, W) in $execution.[[ReadsFrom]]$, (W, R) is in $execution.[[SynchronizesWith]]$ if $R.[[Order]]$ is **SeqCst**, $W.[[Order]]$ is **SeqCst**, and R and W have equal ranges.
- For each element $eventsRecord$ of $execution.[[EventsRecords]]$, the following is true.
 - For each pair (S, Sw) in $eventsRecord.[[AgentSynchronizesWith]]$, (S, Sw) is in $execution.[[SynchronizesWith]]$.
- For each pair (E, D) in $execution.[[HostSynchronizesWith]]$, (E, D) is in $execution.[[SynchronizesWith]]$.

NOTE 1 Owing to convention, write events **synchronizes-with** read events, instead of read events **synchronizes-with** write events.

NOTE 2 **Init** events do not participate in **synchronizes-with**, and are instead constrained directly by **happens-before**.

NOTE 3 Not all **SeqCst** events related by **reads-from** are related by **synchronizes-with**. Only events that also have equal ranges are related by **synchronizes-with**.

NOTE 4 For **Shared Data Block** events R and W such that W **synchronizes-with** R , R may **reads-from** other writes than W .

29.6.6 happens-before

For a **candidate execution** $execution$, $execution.[[HappensBefore]]$ is the least **Relation** on events that satisfies the following.

- For each pair (E, D) in $execution.[[AgentOrder]]$, (E, D) is in $execution.[[HappensBefore]]$.
- For each pair (E, D) in $execution.[[SynchronizesWith]]$, (E, D) is in $execution.[[HappensBefore]]$.
- For each pair (E, D) in $SharedDataBlockEventSet(execution)$, (E, D) is in $execution.[[HappensBefore]]$ if $E.[[Order]]$ is **Init** and E and D have overlapping ranges.
- For each pair (E, D) in $EventSet(execution)$, (E, D) is in $execution.[[HappensBefore]]$ if there is an event F such that the pairs (E, F) and (F, D) are in $execution.[[HappensBefore]]$.

Because happens-before is a superset of [agent-order](#), candidate executions are consistent with the single-thread evaluation semantics of ECMAScript.

29.7 Properties of Valid Executions

29.7.1 Valid Chosen Reads

A [candidate execution](#) *execution* has valid chosen reads if the following abstract operation returns **true**.

1. For each [ReadSharedMemory](#) or [ReadModifyWriteSharedMemory](#) event *R* of [SharedDataBlockEventSet](#)(*execution*), do
 - a. Let *chosenValueRecord* be the element of *execution*.[[ChosenValues]] whose [[Event]] field is *R*.
 - b. Let *chosenValue* be *chosenValueRecord*.[[ChosenValue]].
 - c. Let *readValue* be [ValueOfReadEvent](#)(*execution*, *R*).
 - d. Let *chosenLen* be the number of elements of *chosenValue*.
 - e. Let *readLen* be the number of elements of *readValue*.
 - f. If *chosenLen* ≠ *readLen*, then
 - i. Return **false**.
 - g. If *chosenValue*[*i*] ≠ *readValue*[*i*] for any [integer](#) value *i* in the range 0 through *chosenLen*, exclusive, then
 - i. Return **false**.
2. Return **true**.

29.7.2 Coherent Reads

A [candidate execution](#) *execution* has coherent reads if the following abstract operation returns **true**.

1. For each [ReadSharedMemory](#) or [ReadModifyWriteSharedMemory](#) event *R* of [SharedDataBlockEventSet](#)(*execution*), do
 - a. Let *Ws* be *execution*.[[ReadsBytesFrom]](*R*).
 - b. Let *byteLocation* be *R*.[[ByteIndex]].
 - c. For each element *W* of *Ws*, do
 - i. If (*R*, *W*) is in *execution*.[[HappensBefore]], then
 1. Return **false**.
 - ii. If there is a [WriteSharedMemory](#) or [ReadModifyWriteSharedMemory](#) event *V* that has *byteLocation* in its range such that the pairs (*W*, *V*) and (*V*, *R*) are in *execution*.[[HappensBefore]], then
 1. Return **false**.
 - iii. Set *byteLocation* to *byteLocation* + 1.
2. Return **true**.

29.7.3 Tear Free Reads

A [candidate execution](#) *execution* has tear free reads if the following abstract operation returns **true**.

1. For each [ReadSharedMemory](#) or [ReadModifyWriteSharedMemory](#) event *R* of [SharedDataBlockEventSet](#)(*execution*), do

- a. If $R.[[NoTear]]$ is **true**, then
 - i. **Assert:** The remainder of dividing $R.[[ByteIndex]]$ by $R.[[ElementSize]]$ is 0.
 - ii. For each event W such that (R, W) is in $execution.[[ReadsFrom]]$ and $W.[[NoTear]]$ is **true**, do
 1. If R and W have equal ranges, and there is an event V such that V and W have equal ranges, $V.[[NoTear]]$ is **true**, W is not V , and (R, V) is in $execution.[[ReadsFrom]]$, then
 - a. Return **false**.

2. Return **true**.

NOTE

An event's $[[NoTear]]$ field is **true** when that event was introduced via accessing an **integer** TypedArray, and **false** when introduced via accessing a floating point TypedArray or DataView.

Intuitively, this requirement says when a memory range is accessed in an aligned fashion via an **integer** TypedArray, a single write event on that range must "win" when in a data race with other write events with equal ranges. More precisely, this requirement says an aligned read event cannot read a value composed of bytes from multiple, different write events all with equal ranges. It is possible, however, for an aligned read event to read from multiple write events with overlapping ranges.

29.7.4 Sequentially Consistent Atomics

For a **candidate execution** $execution$, memory-order is a **strict total order** of all events in $EventSet(execution)$ that satisfies the following.

- For each pair (E, D) in $execution.[[HappensBefore]]$, (E, D) is in memory-order.
- For each pair (R, W) in $execution.[[ReadsFrom]]$, there is no **WriteSharedMemory** or **ReadModifyWriteSharedMemory** event V in $SharedDataBlockEventSet(execution)$ such that $V.[[Order]]$ is **SeqCst**, the pairs (W, V) and (V, R) are in memory-order, and any of the following conditions are true.
 - The pair (W, R) is in $execution.[[SynchronizesWith]]$, and V and R have equal ranges.
 - The pairs (W, R) and (V, R) are in $execution.[[HappensBefore]]$, $W.[[Order]]$ is **SeqCst**, and W and V have equal ranges.
 - The pairs (W, R) and (W, V) are in $execution.[[HappensBefore]]$, $R.[[Order]]$ is **SeqCst**, and V and R have equal ranges.

NOTE 1

This clause additionally constrains **SeqCst** events on equal ranges.

- For each **WriteSharedMemory** or **ReadModifyWriteSharedMemory** event W in $SharedDataBlockEventSet(execution)$, if $W.[[Order]]$ is **SeqCst**, then it is not the case that there is an infinite number of **ReadSharedMemory** or **ReadModifyWriteSharedMemory** events in $SharedDataBlockEventSet(execution)$ with equal range that is memory-order before W .

NOTE 2

This clause together with the forward progress guarantee on agents ensure the liveness condition that **SeqCst** writes become visible to **SeqCst** reads with equal range in finite time.

A **candidate execution** has sequentially consistent atomics if a memory-order exists.

NOTE 3 While memory-order includes all events in `EventSet(execution)`, those that are not constrained by `happens-before` or `synchronizes-with` are allowed to occur anywhere in the order.

29.7.5 Valid Executions

A `candidate execution` `execution` is a valid execution (or simply an execution) if all of the following are true.

- The `host` provides a `host-synchronizes-with Relation` for `execution`.`[[HostSynchronizesWith]]`.
- `execution`.`[[HappensBefore]]` is a `strict partial order`.
- `execution` has valid chosen reads.
- `execution` has coherent reads.
- `execution` has tear free reads.
- `execution` has sequentially consistent atomics.

All programs have at least one valid execution.

29.8 Races

For an execution `execution`, two events `E` and `D` in `SharedDataBlockEventSet(execution)` are in a race if the following abstract operation returns `true`.

1. If `E` is not `D`, then
 - a. If the pairs `(E, D)` and `(D, E)` are not in `execution`.`[[HappensBefore]]`, then
 - i. If `E` and `D` are both `WriteSharedMemory` or `ReadModifyWriteSharedMemory` events and `E` and `D` do not have disjoint ranges, then
 1. Return `true`.
 - ii. If either `(E, D)` or `(D, E)` is in `execution`.`[[ReadsFrom]]`, then
 1. Return `true`.
2. Return `false`.

29.9 Data Races

For an execution `execution`, two events `E` and `D` in `SharedDataBlockEventSet(execution)` are in a data race if the following abstract operation returns `true`.

1. If `E` and `D` are in a race in `execution`, then
 - a. If `E`.`[[Order]]` is not `SeqCst` or `D`.`[[Order]]` is not `SeqCst`, then
 - i. Return `true`.
 - b. If `E` and `D` have overlapping ranges, then
 - i. Return `true`.
2. Return `false`.

29.10 Data Race Freedom

An execution `execution` is data race free if there are no two events in `SharedDataBlockEventSet(execution)` that are in a

data race.

A program is data race free if all its executions are data race free.

The [memory model](#) guarantees sequential consistency of all events for data race free programs.

29.11 Shared Memory Guidelines

NOTE 1 The following are guidelines for ECMAScript programmers working with shared memory.

We recommend programs be kept data race free, i.e., make it so that it is impossible for there to be concurrent non-atomic operations on the same memory location. Data race free programs have interleaving semantics where each step in the evaluation semantics of each [agent](#) are interleaved with each other. For data race free programs, it is not necessary to understand the details of the [memory model](#). The details are unlikely to build intuition that will help one to better write ECMAScript.

More generally, even if a program is not data race free it may have predictable behaviour, so long as atomic operations are not involved in any data races and the operations that race all have the same access size. The simplest way to arrange for atomics not to be involved in races is to ensure that different memory cells are used by atomic and non-atomic operations and that atomic accesses of different sizes are not used to access the same cells at the same time. Effectively, the program should treat shared memory as strongly typed as much as possible. One still cannot depend on the ordering and timing of non-atomic accesses that race, but if memory is treated as strongly typed the racing accesses will not "tear" (bits of their values will not be mixed).

NOTE 2 The following are guidelines for ECMAScript implementers writing compiler transformations for programs using shared memory.

It is desirable to allow most program transformations that are valid in a single-[agent](#) setting in a multi-[agent](#) setting, to ensure that the performance of each [agent](#) in a multi-[agent](#) program is as good as it would be in a single-[agent](#) setting. Frequently these transformations are hard to judge. We outline some rules about program transformations that are intended to be taken as normative (in that they are implied by the [memory model](#) or stronger than what the [memory model](#) implies) but which are likely not exhaustive. These rules are intended to apply to program transformations that precede the introductions of the events that make up the [agent-order](#).

Let an *agent-order slice* be the subset of the [agent-order](#) pertaining to a single [agent](#).

Let *possible read values* of a read event be the set of all values of [ValueOfReadEvent](#) for that event across all valid executions.

Any transformation of an agent-order slice that is valid in the absence of shared memory is valid in the presence of shared memory, with the following exceptions.

- *Atomics are carved in stone*: Program transformations must not cause the [SeqCst](#) events in an agent-order slice to be reordered with its [Unordered](#) operations, nor its [SeqCst](#) operations to be reordered with each other, nor may a program transformation remove a [SeqCst](#) operation from the [agent-order](#).

(In practice, the prohibition on reorderings forces a compiler to assume that every SeqCst operation is a synchronization and included in the final [memory-order](#), which it would usually have to assume anyway in the absence of inter-agent program analysis. It also forces the compiler to assume that every call where the callee's effects on the [memory-order](#) are unknown may contain SeqCst operations.)

- *Reads must be stable*: Any given shared memory read must only observe a single value in an execution.

(For example, if what is semantically a single read in the program is executed multiple times then the program is subsequently allowed to observe only one of the values read. A transformation known as rematerialization can violate this rule.)

- *Writes must be stable*: All observable writes to shared memory must follow from program semantics in an execution.

(For example, a transformation may not introduce certain observable writes, such as by using read-modify-write operations on a larger location to write a smaller datum, writing a value to memory that the program could not have written, or writing a just-read value back to the location it was read from, if that location could have been overwritten by another [agent](#) after the read.)

- *Possible read values must be nonempty*: Program transformations cannot cause the possible read values of a shared memory read to become empty.

(Counterintuitively, this rule in effect restricts transformations on writes, because writes have force in [memory model](#) insofar as to be read by read events. For example, writes may be moved and coalesced and sometimes reordered between two SeqCst operations, but the transformation may not remove every write that updates a location; some write must be preserved.)

Examples of transformations that remain valid are: merging multiple non-atomic reads from the same location, reordering non-atomic reads, introducing speculative non-atomic reads, merging multiple non-atomic writes to the same location, reordering non-atomic writes to different locations, and hoisting non-atomic reads out of loops even if that affects termination. Note in general that aliased TypedArrays make it hard to prove that locations are different.

NOTE 3

The following are guidelines for ECMAScript implementers generating machine code for shared memory accesses.

For architectures with memory models no weaker than those of ARM or Power, non-atomic stores and loads may be compiled to bare stores and loads on the target architecture. Atomic stores and loads may be compiled down to instructions that guarantee sequential consistency. If no such instructions exist, memory barriers are to be employed, such as placing barriers on both sides of a bare store or load. Read-modify-write operations may be compiled to read-modify-write instructions on the target architecture, such as **LOCK**-prefixed instructions on x86, load-exclusive/store-exclusive instructions on ARM, and load-link/store-conditional instructions on Power.

Specifically, the [memory model](#) is intended to allow code generation as follows.

- Every atomic operation in the program is assumed to be necessary.
- Atomic operations are never rearranged with each other or with non-atomic operations.
- Functions are always assumed to perform atomic operations.
- Atomic operations are never implemented as read-modify-write operations on larger data, but as non-lock-free atomics if the platform does not have atomic operations of the appropriate size. (We already assume that every platform has normal memory access operations of every interesting size.)

Naive code generation uses these patterns:

- Regular loads and stores compile to single load and store instructions.
- Lock-free atomic loads and stores compile to a full (sequentially consistent) fence, a regular load or store, and a full fence.
- Lock-free atomic read-modify-write accesses compile to a full fence, an atomic read-modify-write instruction sequence, and a full fence.
- Non-lock-free atomics compile to a spinlock acquire, a full fence, a series of non-atomic load and store instructions, a full fence, and a spinlock release.

That mapping is correct so long as an atomic operation on an address range does not race with a non-atomic write or with an atomic operation of different size. However, that is all we need: the [memory model](#) effectively demotes the atomic operations involved in a race to non-atomic status. On the other hand, the naive mapping is quite strong: it allows atomic operations to be used as sequentially consistent fences, which the [memory model](#) does not actually guarantee.

A number of local improvements to those basic patterns are also intended to be legal:

- There are obvious platform-dependent improvements that remove redundant fences. For example, on x86 the fences around lock-free atomic loads and stores can always be omitted except for the fence following a store, and no fence is needed for lock-free read-modify-write instructions, as these all use LOCK-prefixed instructions. On many platforms there are fences of several strengths, and weaker fences can be used in certain contexts without destroying sequential consistency.
- Most modern platforms support lock-free atomics for all the data sizes required by ECMAScript atomics. Should non-lock-free atomics be needed, the fences surrounding the body of the atomic operation can usually be folded into the lock and unlock steps. The simplest solution for non-lock-free atomics is to have a single lock word per `SharedArrayBuffer`.
- There are also more complicated platform-dependent local improvements, requiring some code analysis. For example, two back-to-back fences often have the same effect as a single fence, so if code is generated for two atomic operations in sequence, only a single fence need separate them. On x86, even a single fence separating atomic stores can be omitted, as the fence following a store is only needed to separate the store from a subsequent load.

A Grammar Summary

A.1 Lexical Grammar

SourceCharacter ::

any Unicode code point

InputElementDiv ::

WhiteSpace

LineTerminator

Comment

CommonToken

DivPunctuator

RightBracePunctuator

InputElementRegExp ::

WhiteSpace

LineTerminator

Comment

CommonToken

RightBracePunctuator

RegularExpressionLiteral

InputElementRegExpOrTemplateTail ::

WhiteSpace

LineTerminator

Comment

CommonToken

RegularExpressionLiteral

TemplateSubstitutionTail

InputElementTemplateTail ::

WhiteSpace

LineTerminator

Comment

CommonToken

DivPunctuator

TemplateSubstitutionTail

WhiteSpace ::

<TAB>

<VT>

<FF>

<SP>

<NBSP>

<ZWNBSP>

<USP>

LineTerminator ::

<LF>

<CR>

<LS>

<PS>

LineTerminatorSequence ::

- <LF>
- <CR> [lookahead ≠ <LF>]
- <LS>
- <PS>
- <CR> <LF>

Comment ::

- MultiLineComment*
- SingleLineComment*

MultiLineComment ::

- /* MultiLineCommentChars_{opt} */*

MultiLineCommentChars ::

- MultiLineNotAsteriskChar MultiLineCommentChars_{opt}*
- * PostAsteriskCommentChars_{opt}*

PostAsteriskCommentChars ::

- MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars_{opt}*
- * PostAsteriskCommentChars_{opt}*

MultiLineNotAsteriskChar ::

- SourceCharacter* but not ***

MultiLineNotForwardSlashOrAsteriskChar ::

- SourceCharacter* but not one of / or ***

SingleLineComment ::

- // SingleLineCommentChars_{opt}*

SingleLineCommentChars ::

- SingleLineCommentChar SingleLineCommentChars_{opt}*

SingleLineCommentChar ::

- SourceCharacter* but not *LineTerminator*

CommonToken ::

- IdentifierName*
- Punctuator*
- NumericLiteral*
- StringLiteral*
- Template*

IdentifierName ::

- IdentifierStart*
- IdentifierName IdentifierPart*

IdentifierStart ::

- UnicodeIDStart*
- \$*
-
- \ UnicodeEscapeSequence*

IdentifierPart ::

- UnicodeIDContinue*
- \$*

\ UnicodeEscapeSequence

<ZWNJ>

<ZWJ>

UnicodeIDStart ::

any Unicode code point with the Unicode property “ID_Start”

UnicodeIDContinue ::

any Unicode code point with the Unicode property “ID_Continue”

ReservedWord :: **one of**

**await break case catch class const continue debugger default delete do else enum export extends false
finally for function if import in instanceof new null return super switch this throw true try
typeof var void while with yield**

Punctuator ::

OptionalChainingPunctuator

OtherPunctuator

OptionalChainingPunctuator ::

?. [lookahead \notin *DecimalDigit*]

OtherPunctuator :: **one of**

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

DivPunctuator ::

/

/=

RightBracePunctuator ::

}

NullLiteral ::

null

BooleanLiteral ::

true

false

NumericLiteralSeparator ::

-

NumericLiteral ::

DecimalLiteral

DecimalBigIntegerLiteral

NonDecimalIntegerLiteral *[+Sep]*

NonDecimalIntegerLiteral *[+Sep]* *BigIntLiteralSuffix*

DecimalBigIntegerLiteral ::

0 *BigIntLiteralSuffix*

NonZeroDigit *DecimalDigits* *[+Sep]* *opt* *BigIntLiteralSuffix*

NonZeroDigit *NumericLiteralSeparator* *DecimalDigits* *[+Sep]* *BigIntLiteralSuffix*

NonDecimalIntegerLiteral *[Sep]* ::

BinaryIntegerLiteral *[?Sep]*

OctalIntegerLiteral *[?Sep]*

HexIntegerLiteral *[?Sep]*

BigIntLiteralSuffix ::

n

DecimalLiteral ::

DecimalIntegerLiteral . *DecimalDigits*_[+Sep] *opt* *ExponentPart*_[+Sep] *opt*

. *DecimalDigits*_[+Sep] *ExponentPart*_[+Sep] *opt*

DecimalIntegerLiteral *ExponentPart*_[+Sep] *opt*

DecimalIntegerLiteral ::

0

NonZeroDigit

NonZeroDigit *NumericLiteralSeparator*_{opt} *DecimalDigits*_[+Sep]

*DecimalDigits*_[Sep] ::

DecimalDigit

*DecimalDigits*_[?Sep] *DecimalDigit*

[+Sep] *DecimalDigits*[+Sep] *NumericLiteralSeparator* *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*_[Sep] ::

ExponentIndicator *SignedInteger*_[?Sep]

ExponentIndicator :: **one of**

e E

*SignedInteger*_[Sep] ::

*DecimalDigits*_[?Sep]

+ *DecimalDigits*_[?Sep]

- *DecimalDigits*_[?Sep]

*BinaryIntegerLiteral*_[Sep] ::

0b *BinaryDigits*_[?Sep]

0B *BinaryDigits*_[?Sep]

*BinaryDigits*_[Sep] ::

BinaryDigit

*BinaryDigits*_[?Sep] *BinaryDigit*

[+Sep] *BinaryDigits*[+Sep] *NumericLiteralSeparator* *BinaryDigit*

BinaryDigit :: **one of**

0 1

*OctalIntegerLiteral*_[Sep] ::

0o *OctalDigits*_[?Sep]

0O *OctalDigits*_[?Sep]

*OctalDigits*_[Sep] ::

OctalDigit

*OctalDigits*_[?Sep] *OctalDigit*

[+Sep] *OctalDigits*[+Sep] *NumericLiteralSeparator* *OctalDigit*

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

*HexIntegerLiteral*_[Sep] ::
0x *HexDigits*_[?Sep]
0X *HexDigits*_[?Sep]

*HexDigits*_[Sep] ::
HexDigit
*HexDigits*_[?Sep] *HexDigit*
[+Sep] *HexDigits*[+Sep] *NumericLiteralSeparator* *HexDigit*

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

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

DoubleStringCharacters ::
DoubleStringCharacter *DoubleStringCharacters*_{opt}

SingleStringCharacters ::
SingleStringCharacter *SingleStringCharacters*_{opt}

DoubleStringCharacter ::
SourceCharacter but not one of " or \ or *LineTerminator*
<LS>
<PS>
\ *EscapeSequence*
LineContinuation

SingleStringCharacter ::
SourceCharacter but not one of ' or \ or *LineTerminator*
<LS>
<PS>
\ *EscapeSequence*
LineContinuation

LineContinuation ::
\ *LineTerminatorSequence*

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

CharacterEscapeSequence ::
SingleEscapeCharacter
NonEscapeCharacter

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

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

EscapeCharacter ::
 SingleEscapeCharacter
 DecimalDigit
 x
 u

HexEscapeSequence ::
 x *HexDigit* *HexDigit*

UnicodeEscapeSequence ::
 u *Hex4Digits*
 u{ *CodePoint* }

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 \ 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*

Template ::
 NoSubstitutionTemplate
 TemplateHead

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

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

TemplateSubstitutionTail ::
TemplateMiddle
TemplateTail

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

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

TemplateCharacters ::
TemplateCharacter *TemplateCharacters*_{opt}

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

NotEscapeSequence ::
0 *DecimalDigit*
DecimalDigit but not **0**
x [**lookahead** ∉ *HexDigit*]
x *HexDigit* [**lookahead** ∉ *HexDigit*]
u [**lookahead** ∉ *HexDigit*] [**lookahead** ≠ **{**]
u *HexDigit* [**lookahead** ∉ *HexDigit*]
u *HexDigit* *HexDigit* [**lookahead** ∉ *HexDigit*]
u *HexDigit* *HexDigit* *HexDigit* [**lookahead** ∉ *HexDigit*]
u **{** [**lookahead** ∉ *HexDigit*]
u **{** *NotCodePoint* [**lookahead** ∉ *HexDigit*]
u **{** *CodePoint* [**lookahead** ∉ *HexDigit*] [**lookahead** ≠ **}**]

NotCodePoint ::
*HexDigits*_[~Sep] but only if MV of *HexDigits* > 0x10FFFF

CodePoint ::
*HexDigits*_[~Sep] but only if MV of *HexDigits* ≤ 0x10FFFF

A.2 Expressions

*IdentifierReference*_[Yield, Await] :
Identifier
 [~Yield] **yield**
 [~Await] **await**

*BindingIdentifier*_[Yield, Await] :

Identifier

yield

await

LabelIdentifier [Yield, Await] :

Identifier

[~Yield] **yield**

[~Await] **await**

Identifier :

IdentifierName but not *ReservedWord*

PrimaryExpression [Yield, Await] :

this

IdentifierReference [?Yield, ?Await]

Literal

ArrayLiteral [?Yield, ?Await]

ObjectLiteral [?Yield, ?Await]

FunctionExpression

ClassExpression [?Yield, ?Await]

GeneratorExpression

AsyncFunctionExpression

AsyncGeneratorExpression

RegularExpressionLiteral

TemplateLiteral [?Yield, ?Await, ~Tagged]

CoverParenthesizedExpressionAndArrowParameterList [?Yield, ?Await]

CoverParenthesizedExpressionAndArrowParameterList [Yield, Await] :

(*Expression* [+In, ?Yield, ?Await])

(*Expression* [+In, ?Yield, ?Await] ,)

()

(... *BindingIdentifier* [?Yield, ?Await])

(... *BindingPattern* [?Yield, ?Await])

(*Expression* [+In, ?Yield, ?Await] , ... *BindingIdentifier* [?Yield, ?Await])

(*Expression* [+In, ?Yield, ?Await] , ... *BindingPattern* [?Yield, ?Await])

When processing an instance of the production

PrimaryExpression [Yield, Await] : *CoverParenthesizedExpressionAndArrowParameterList* [?Yield, ?Await]

the interpretation of *CoverParenthesizedExpressionAndArrowParameterList* is refined using the following grammar:

ParenthesizedExpression [Yield, Await] :

(*Expression* [+In, ?Yield, ?Await])

Literal :

NullLiteral

BooleanLiteral

```

    NumericLiteral
    StringLiteral
ArrayLiteral[Yield, Await] :
    [ Elisionopt ]
    [ ElementList[?Yield, ?Await] ]
    [ ElementList[?Yield, ?Await] , Elisionopt ]
ElementList[Yield, Await] :
    Elisionopt AssignmentExpression[+In, ?Yield, ?Await]
    Elisionopt SpreadElement[?Yield, ?Await]
    ElementList[?Yield, ?Await] , Elisionopt AssignmentExpression[+In, ?Yield, ?Await]
    ElementList[?Yield, ?Await] , Elisionopt SpreadElement[?Yield, ?Await]
Elision :
    ,
    Elision ,
SpreadElement[Yield, Await] :
    ... AssignmentExpression[+In, ?Yield, ?Await]
ObjectLiteral[Yield, Await] :
    { }
    { PropertyDefinitionList[?Yield, ?Await] }
    { PropertyDefinitionList[?Yield, ?Await] , }
PropertyDefinitionList[Yield, Await] :
    PropertyDefinition[?Yield, ?Await]
    PropertyDefinitionList[?Yield, ?Await] , PropertyDefinition[?Yield, ?Await]
PropertyDefinition[Yield, Await] :
    IdentifierReference[?Yield, ?Await]
    CoverInitializedName[?Yield, ?Await]
    PropertyName[?Yield, ?Await] : AssignmentExpression[+In, ?Yield, ?Await]
    MethodDefinition[?Yield, ?Await]
    ... AssignmentExpression[+In, ?Yield, ?Await]
PropertyName[Yield, Await] :
    LiteralPropertyName
    ComputedPropertyName[?Yield, ?Await]
LiteralPropertyName :
    IdentifierName
    StringLiteral
    NumericLiteral
ComputedPropertyName[Yield, Await] :
    [ AssignmentExpression[+In, ?Yield, ?Await] ]
CoverInitializedName[Yield, Await] :
    IdentifierReference[?Yield, ?Await] Initializer[+In, ?Yield, ?Await]
Initializer[In, Yield, Await] :

```

```

    = AssignmentExpression[?In, ?Yield, ?Await]
TemplateLiteral[Yield, Await, Tagged] :
    NoSubstitutionTemplate
    SubstitutionTemplate[?Yield, ?Await, ?Tagged]
SubstitutionTemplate[Yield, Await, Tagged] :
    TemplateHead Expression[+In, ?Yield, ?Await] TemplateSpans[?Yield, ?Await, ?Tagged]
TemplateSpans[Yield, Await, Tagged] :
    TemplateTail
    TemplateMiddleList[?Yield, ?Await, ?Tagged] TemplateTail
TemplateMiddleList[Yield, Await, Tagged] :
    TemplateMiddle Expression[+In, ?Yield, ?Await]
    TemplateMiddleList[?Yield, ?Await, ?Tagged] TemplateMiddle Expression[+In, ?Yield, ?Await]
MemberExpression[Yield, Await] :
    PrimaryExpression[?Yield, ?Await]
    MemberExpression[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    MemberExpression[?Yield, ?Await] . IdentifierName
    MemberExpression[?Yield, ?Await] TemplateLiteral[?Yield, ?Await, +Tagged]
    SuperProperty[?Yield, ?Await]
    MetaProperty
    new MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]
SuperProperty[Yield, Await] :
    super [ Expression[+In, ?Yield, ?Await] ]
    super . IdentifierName
MetaProperty :
    NewTarget
    ImportMeta
NewTarget :
    new . target
ImportMeta :
    import . meta
NewExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await]
    new NewExpression[?Yield, ?Await]
CallExpression[Yield, Await] :
    CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await]
    SuperCall[?Yield, ?Await]
    ImportCall[?Yield, ?Await]
    CallExpression[?Yield, ?Await] Arguments[?Yield, ?Await]
    CallExpression[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    CallExpression[?Yield, ?Await] . IdentifierName
    CallExpression[?Yield, ?Await] TemplateLiteral[?Yield, ?Await, +Tagged]

```

When processing an instance of the production

*CallExpression*_[Yield, Await] : *CoverCallExpressionAndAsyncArrowHead*_[?Yield, ?Await]

the interpretation of *CoverCallExpressionAndAsyncArrowHead* is refined using the following grammar:

*CallMemberExpression*_[Yield, Await] :

*MemberExpression*_[?Yield, ?Await] *Arguments*_[?Yield, ?Await]

*SuperCall*_[Yield, Await] :

super *Arguments*_[?Yield, ?Await]

*ImportCall*_[Yield, Await] :

import (*AssignmentExpression*_[+In, ?Yield, ?Await])

*Arguments*_[Yield, Await] :

()

(*ArgumentList*_[?Yield, ?Await])

(*ArgumentList*_[?Yield, ?Await] ,)

*ArgumentList*_[Yield, Await] :

*AssignmentExpression*_[+In, ?Yield, ?Await]

... *AssignmentExpression*_[+In, ?Yield, ?Await]

*ArgumentList*_[?Yield, ?Await] , *AssignmentExpression*_[+In, ?Yield, ?Await]

*ArgumentList*_[?Yield, ?Await] , ... *AssignmentExpression*_[+In, ?Yield, ?Await]

*OptionalExpression*_[Yield, Await] :

*MemberExpression*_[?Yield, ?Await] *OptionalChain*_[?Yield, ?Await]

*CallExpression*_[?Yield, ?Await] *OptionalChain*_[?Yield, ?Await]

*OptionalExpression*_[?Yield, ?Await] *OptionalChain*_[?Yield, ?Await]

*OptionalChain*_[Yield, Await] :

? . *Arguments*_[?Yield, ?Await]

? . [*Expression*_[+In, ?Yield, ?Await]]

? . *IdentifierName*

? . *TemplateLiteral*_[?Yield, ?Await, +Tagged]

*OptionalChain*_[?Yield, ?Await] *Arguments*_[?Yield, ?Await]

*OptionalChain*_[?Yield, ?Await] [*Expression*_[+In, ?Yield, ?Await]]

*OptionalChain*_[?Yield, ?Await] . *IdentifierName*

*OptionalChain*_[?Yield, ?Await] *TemplateLiteral*_[?Yield, ?Await, +Tagged]

*LeftHandSideExpression*_[Yield, Await] :

*NewExpression*_[?Yield, ?Await]

*CallExpression*_[?Yield, ?Await]

*OptionalExpression*_[?Yield, ?Await]

*UpdateExpression*_[Yield, Await] :

*LeftHandSideExpression*_[?Yield, ?Await]

*LeftHandSideExpression*_[?Yield, ?Await] [no *LineTerminator* here] ++

```

LeftHandSideExpression [?Yield, ?Await] [no LineTerminator here] --
++ UnaryExpression [?Yield, ?Await]
-- UnaryExpression [?Yield, ?Await]
UnaryExpression [Yield, Await] :
    UpdateExpression [?Yield, ?Await]
    delete UnaryExpression [?Yield, ?Await]
    void UnaryExpression [?Yield, ?Await]
    typeof UnaryExpression [?Yield, ?Await]
    + UnaryExpression [?Yield, ?Await]
    - UnaryExpression [?Yield, ?Await]
    ~ UnaryExpression [?Yield, ?Await]
    ! UnaryExpression [?Yield, ?Await]
    [+Await] AwaitExpression [?Yield]
ExponentiationExpression [Yield, Await] :
    UnaryExpression [?Yield, ?Await]
    UpdateExpression [?Yield, ?Await] ** ExponentiationExpression [?Yield, ?Await]
MultiplicativeExpression [Yield, Await] :
    ExponentiationExpression [?Yield, ?Await]
    MultiplicativeExpression [?Yield, ?Await] MultiplicativeOperator ExponentiationExpression [?Yield, ?Await]
MultiplicativeOperator : one of
    * / %
AdditiveExpression [Yield, Await] :
    MultiplicativeExpression [?Yield, ?Await]
    AdditiveExpression [?Yield, ?Await] + MultiplicativeExpression [?Yield, ?Await]
    AdditiveExpression [?Yield, ?Await] - MultiplicativeExpression [?Yield, ?Await]
ShiftExpression [Yield, Await] :
    AdditiveExpression [?Yield, ?Await]
    ShiftExpression [?Yield, ?Await] << AdditiveExpression [?Yield, ?Await]
    ShiftExpression [?Yield, ?Await] >> AdditiveExpression [?Yield, ?Await]
    ShiftExpression [?Yield, ?Await] >>> AdditiveExpression [?Yield, ?Await]
RelationalExpression [In, Yield, Await] :
    ShiftExpression [?Yield, ?Await]
    RelationalExpression [?In, ?Yield, ?Await] < ShiftExpression [?Yield, ?Await]
    RelationalExpression [?In, ?Yield, ?Await] > ShiftExpression [?Yield, ?Await]
    RelationalExpression [?In, ?Yield, ?Await] <= ShiftExpression [?Yield, ?Await]
    RelationalExpression [?In, ?Yield, ?Await] >= ShiftExpression [?Yield, ?Await]
    RelationalExpression [?In, ?Yield, ?Await] instanceof ShiftExpression [?Yield, ?Await]
    [+In] RelationalExpression [+In, ?Yield, ?Await] in ShiftExpression [?Yield, ?Await]
EqualityExpression [In, Yield, Await] :
    RelationalExpression [?In, ?Yield, ?Await]

```

```

EqualityExpression[?In, ?Yield, ?Await] == RelationalExpression[?In, ?Yield, ?Await]
EqualityExpression[?In, ?Yield, ?Await] != RelationalExpression[?In, ?Yield, ?Await]
EqualityExpression[?In, ?Yield, ?Await] === RelationalExpression[?In, ?Yield, ?Await]
EqualityExpression[?In, ?Yield, ?Await] !== RelationalExpression[?In, ?Yield, ?Await]
BitwiseANDExpression[In, Yield, Await] :
    EqualityExpression[?In, ?Yield, ?Await]
    BitwiseANDExpression[?In, ?Yield, ?Await] & EqualityExpression[?In, ?Yield, ?Await]
BitwiseXORExpression[In, Yield, Await] :
    BitwiseANDExpression[?In, ?Yield, ?Await]
    BitwiseXORExpression[?In, ?Yield, ?Await] ^ BitwiseANDExpression[?In, ?Yield, ?Await]
BitwiseORExpression[In, Yield, Await] :
    BitwiseXORExpression[?In, ?Yield, ?Await]
    BitwiseORExpression[?In, ?Yield, ?Await] | BitwiseXORExpression[?In, ?Yield, ?Await]
LogicalANDExpression[In, Yield, Await] :
    BitwiseORExpression[?In, ?Yield, ?Await]
    LogicalANDExpression[?In, ?Yield, ?Await] && BitwiseORExpression[?In, ?Yield, ?Await]
LogicalORExpression[In, Yield, Await] :
    LogicalANDExpression[?In, ?Yield, ?Await]
    LogicalORExpression[?In, ?Yield, ?Await] || LogicalANDExpression[?In, ?Yield, ?Await]
CoalesceExpression[In, Yield, Await] :
    CoalesceExpressionHead[?In, ?Yield, ?Await] ?? BitwiseORExpression[?In, ?Yield, ?Await]
CoalesceExpressionHead[In, Yield, Await] :
    CoalesceExpression[?In, ?Yield, ?Await]
    BitwiseORExpression[?In, ?Yield, ?Await]
ShortCircuitExpression[In, Yield, Await] :
    LogicalORExpression[?In, ?Yield, ?Await]
    CoalesceExpression[?In, ?Yield, ?Await]
ConditionalExpression[In, Yield, Await] :
    ShortCircuitExpression[?In, ?Yield, ?Await]
    ShortCircuitExpression[?In, ?Yield, ?Await] ? AssignmentExpression[+In, ?Yield, ?Await] :
        AssignmentExpression[?In, ?Yield, ?Await]
AssignmentExpression[In, Yield, Await] :
    ConditionalExpression[?In, ?Yield, ?Await]
    [+Yield] YieldExpression[?In, ?Await]
    ArrowFunction[?In, ?Yield, ?Await]
    AsyncArrowFunction[?In, ?Yield, ?Await]
    LeftHandSideExpression[?Yield, ?Await] = AssignmentExpression[?In, ?Yield, ?Await]
    LeftHandSideExpression[?Yield, ?Await] AssignmentOperator AssignmentExpression[?In, ?Yield, ?Await]
    LeftHandSideExpression[?Yield, ?Await] &&= AssignmentExpression[?In, ?Yield, ?Await]
    LeftHandSideExpression[?Yield, ?Await] ||= AssignmentExpression[?In, ?Yield, ?Await]

```

LeftHandSideExpression [*?Yield*, *?Await*] *??= AssignmentExpression* [*?In*, *?Yield*, *?Await*]

AssignmentOperator : **one of**

**= /= %= += -= <<= >>= >>>= &= ^= |= **=*

In certain circumstances when processing an instance of the production

AssignmentExpression [*In*, *Yield*, *Await*] : *LeftHandSideExpression* [*?Yield*, *?Await*] =
AssignmentExpression [*?In*, *?Yield*, *?Await*]

the interpretation of *LeftHandSideExpression* is refined using the following grammar:

AssignmentPattern [*Yield*, *Await*] :

ObjectAssignmentPattern [*?Yield*, *?Await*]

ArrayAssignmentPattern [*?Yield*, *?Await*]

ObjectAssignmentPattern [*Yield*, *Await*] :

{ }

{ *AssignmentRestProperty* [*?Yield*, *?Await*] }

{ *AssignmentPropertyList* [*?Yield*, *?Await*] }

{ *AssignmentPropertyList* [*?Yield*, *?Await*] , *AssignmentRestProperty* [*?Yield*, *?Await*] *opt* }

ArrayAssignmentPattern [*Yield*, *Await*] :

[*Elision* *opt* *AssignmentRestElement* [*?Yield*, *?Await*] *opt*]

[*AssignmentElementList* [*?Yield*, *?Await*]]

[*AssignmentElementList* [*?Yield*, *?Await*] , *Elision* *opt* *AssignmentRestElement* [*?Yield*, *?Await*] *opt*]

AssignmentRestProperty [*Yield*, *Await*] :

... *DestructuringAssignmentTarget* [*?Yield*, *?Await*]

AssignmentPropertyList [*Yield*, *Await*] :

AssignmentProperty [*?Yield*, *?Await*]

AssignmentPropertyList [*?Yield*, *?Await*] , *AssignmentProperty* [*?Yield*, *?Await*]

AssignmentElementList [*Yield*, *Await*] :

AssignmentElisionElement [*?Yield*, *?Await*]

AssignmentElementList [*?Yield*, *?Await*] , *AssignmentElisionElement* [*?Yield*, *?Await*]

AssignmentElisionElement [*Yield*, *Await*] :

Elision *opt* *AssignmentElement* [*?Yield*, *?Await*]

AssignmentProperty [*Yield*, *Await*] :

IdentifierReference [*?Yield*, *?Await*] *Initializer* [*+In*, *?Yield*, *?Await*] *opt*

PropertyName [*?Yield*, *?Await*] : *AssignmentElement* [*?Yield*, *?Await*]

AssignmentElement [*Yield*, *Await*] :

DestructuringAssignmentTarget [*?Yield*, *?Await*] *Initializer* [*+In*, *?Yield*, *?Await*] *opt*

AssignmentRestElement [*Yield*, *Await*] :

... *DestructuringAssignmentTarget* [*?Yield*, *?Await*]

DestructuringAssignmentTarget [*Yield*, *Await*] :

LeftHandSideExpression [*?Yield*, *?Await*]

*Expression*_[In, Yield, Await] :
 *AssignmentExpression*_[?In, ?Yield, ?Await]
 *Expression*_[?In, ?Yield, ?Await] . *AssignmentExpression*_[?In, ?Yield, ?Await]

A.3 Statements

*Statement*_[Yield, Await, Return] :
 *BlockStatement*_[?Yield, ?Await, ?Return]
 *VariableStatement*_[?Yield, ?Await]
 EmptyStatement
 *ExpressionStatement*_[?Yield, ?Await]
 *IfStatement*_[?Yield, ?Await, ?Return]
 *BreakableStatement*_[?Yield, ?Await, ?Return]
 *ContinueStatement*_[?Yield, ?Await]
 *BreakStatement*_[?Yield, ?Await]
 [+Return] *ReturnStatement*_[?Yield, ?Await]
 *WithStatement*_[?Yield, ?Await, ?Return]
 *LabelledStatement*_[?Yield, ?Await, ?Return]
 *ThrowStatement*_[?Yield, ?Await]
 *TryStatement*_[?Yield, ?Await, ?Return]
 DebuggerStatement

*Declaration*_[Yield, Await] :
 *HoistableDeclaration*_[?Yield, ?Await, ~Default]
 *ClassDeclaration*_[?Yield, ?Await, ~Default]
 *LexicalDeclaration*_[+In, ?Yield, ?Await]

*HoistableDeclaration*_[Yield, Await, Default] :
 *FunctionDeclaration*_[?Yield, ?Await, ?Default]
 *GeneratorDeclaration*_[?Yield, ?Await, ?Default]
 *AsyncFunctionDeclaration*_[?Yield, ?Await, ?Default]
 *AsyncGeneratorDeclaration*_[?Yield, ?Await, ?Default]

*BreakableStatement*_[Yield, Await, Return] :
 *IterationStatement*_[?Yield, ?Await, ?Return]
 *SwitchStatement*_[?Yield, ?Await, ?Return]

*BlockStatement*_[Yield, Await, Return] :
 *Block*_[?Yield, ?Await, ?Return]

*Block*_[Yield, Await, Return] :
 { *StatementList*_[?Yield, ?Await, ?Return] *opt* }

*StatementList*_[Yield, Await, Return] :
 *StatementListItem*_[?Yield, ?Await, ?Return]
 *StatementList*_[?Yield, ?Await, ?Return] *StatementListItem*_[?Yield, ?Await, ?Return]

*StatementListItem*_[Yield, Await, Return] :

```

    Statement[?Yield, ?Await, ?Return]
    Declaration[?Yield, ?Await]
LexicalDeclaration[In, Yield, Await] :
    LetOrConst BindingList[?In, ?Yield, ?Await] ;
LetOrConst :
    let
    const
BindingList[In, Yield, Await] :
    LexicalBinding[?In, ?Yield, ?Await]
    BindingList[?In, ?Yield, ?Await] , LexicalBinding[?In, ?Yield, ?Await]
LexicalBinding[In, Yield, Await] :
    BindingIdentifier[?Yield, ?Await] Initializer[?In, ?Yield, ?Await] opt
    BindingPattern[?Yield, ?Await] Initializer[?In, ?Yield, ?Await]
VariableStatement[Yield, Await] :
    var VariableDeclarationList[+In, ?Yield, ?Await] ;
VariableDeclarationList[In, Yield, Await] :
    VariableDeclaration[?In, ?Yield, ?Await]
    VariableDeclarationList[?In, ?Yield, ?Await] , VariableDeclaration[?In, ?Yield, ?Await]
VariableDeclaration[In, Yield, Await] :
    BindingIdentifier[?Yield, ?Await] Initializer[?In, ?Yield, ?Await] opt
    BindingPattern[?Yield, ?Await] Initializer[?In, ?Yield, ?Await]
BindingPattern[Yield, Await] :
    ObjectBindingPattern[?Yield, ?Await]
    ArrayBindingPattern[?Yield, ?Await]
ObjectBindingPattern[Yield, Await] :
    { }
    { BindingRestProperty[?Yield, ?Await] }
    { BindingPropertyList[?Yield, ?Await] }
    { BindingPropertyList[?Yield, ?Await] , BindingRestProperty[?Yield, ?Await] opt }
ArrayBindingPattern[Yield, Await] :
    [ Elisionopt BindingRestElement[?Yield, ?Await] opt ]
    [ BindingElementList[?Yield, ?Await] ]
    [ BindingElementList[?Yield, ?Await] , Elisionopt BindingRestElement[?Yield, ?Await] opt ]
BindingRestProperty[Yield, Await] :
    ... BindingIdentifier[?Yield, ?Await]
BindingPropertyList[Yield, Await] :
    BindingProperty[?Yield, ?Await]
    BindingPropertyList[?Yield, ?Await] , BindingProperty[?Yield, ?Await]
BindingElementList[Yield, Await] :
    BindingElisionElement[?Yield, ?Await]
    BindingElementList[?Yield, ?Await] , BindingElisionElement[?Yield, ?Await]

```

```

BindingElisionElement [Yield, Await] :
    Elisionopt BindingElement [?Yield, ?Await]
BindingProperty [Yield, Await] :
    SingleNameBinding [?Yield, ?Await]
    PropertyName [?Yield, ?Await] : BindingElement [?Yield, ?Await]
BindingElement [Yield, Await] :
    SingleNameBinding [?Yield, ?Await]
    BindingPattern [?Yield, ?Await] Initializer [+In, ?Yield, ?Await] opt
SingleNameBinding [Yield, Await] :
    BindingIdentifier [?Yield, ?Await] Initializer [+In, ?Yield, ?Await] opt
BindingRestElement [Yield, Await] :
    ... BindingIdentifier [?Yield, ?Await]
    ... BindingPattern [?Yield, ?Await]
EmptyStatement :
    ;
ExpressionStatement [Yield, Await] :
    [lookahead ∉ { {, function, async [no LineTerminator here] function, class, let [ ] } ]
    Expression [+In, ?Yield, ?Await] ;
IfStatement [Yield, Await, Return] :
    if ( Expression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return] else
        Statement [?Yield, ?Await, ?Return]
    if ( Expression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return] [lookahead ≠ else]
IterationStatement [Yield, Await, Return] :
    DoWhileStatement [?Yield, ?Await, ?Return]
    WhileStatement [?Yield, ?Await, ?Return]
    ForStatement [?Yield, ?Await, ?Return]
    ForInOfStatement [?Yield, ?Await, ?Return]
DoWhileStatement [Yield, Await, Return] :
    do Statement [?Yield, ?Await, ?Return] while ( Expression [+In, ?Yield, ?Await] ) ;
WhileStatement [Yield, Await, Return] :
    while ( Expression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return]
ForStatement [Yield, Await, Return] :
    for ( [lookahead ≠ let [ ] Expression [~In, ?Yield, ?Await] opt ; Expression [+In, ?Yield, ?Await] opt ;
        Expression [+In, ?Yield, ?Await] opt ) Statement [?Yield, ?Await, ?Return]
    for ( var VariableDeclarationList [~In, ?Yield, ?Await] ; Expression [+In, ?Yield, ?Await] opt ;
        Expression [+In, ?Yield, ?Await] opt ) Statement [?Yield, ?Await, ?Return]
    for ( LexicalDeclaration [~In, ?Yield, ?Await] Expression [+In, ?Yield, ?Await] opt ;
        Expression [+In, ?Yield, ?Await] opt ) Statement [?Yield, ?Await, ?Return]
ForInOfStatement [Yield, Await, Return] :
    for ( [lookahead ≠ let [ ] LeftHandSideExpression [?Yield, ?Await] in Expression [+In, ?Yield, ?Await] )
        Statement [?Yield, ?Await, ?Return]

```

```

for ( var ForBinding[?Yield, ?Await] in Expression[+In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
for ( ForDeclaration[?Yield, ?Await] in Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
for ( [lookahead ∉ { let, async of } ] LeftHandSideExpression[?Yield, ?Await] of
    AssignmentExpression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
for ( var ForBinding[?Yield, ?Await] of AssignmentExpression[+In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
for ( ForDeclaration[?Yield, ?Await] of AssignmentExpression[+In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
[+Await] for await ( [lookahead ≠ let] LeftHandSideExpression[?Yield, ?Await] of
    AssignmentExpression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
[+Await] for await ( var ForBinding[?Yield, ?Await] of AssignmentExpression[+In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
[+Await] for await ( ForDeclaration[?Yield, ?Await] of AssignmentExpression[+In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
ForDeclaration[Yield, Await] :
    LetOrConst ForBinding[?Yield, ?Await]
ForBinding[Yield, Await] :
    BindingIdentifier[?Yield, ?Await]
    BindingPattern[?Yield, ?Await]
ContinueStatement[Yield, Await] :
    continue ;
    continue [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;
BreakStatement[Yield, Await] :
    break ;
    break [no LineTerminator here] LabelIdentifier[?Yield, ?Await] ;
ReturnStatement[Yield, Await] :
    return ;
    return [no LineTerminator here] Expression[+In, ?Yield, ?Await] ;
WithStatement[Yield, Await, Return] :
    with ( Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
SwitchStatement[Yield, Await, Return] :
    switch ( Expression[+In, ?Yield, ?Await] ) CaseBlock[?Yield, ?Await, ?Return]
CaseBlock[Yield, Await, Return] :
    { CaseClauses[?Yield, ?Await, ?Return] opt }
    { CaseClauses[?Yield, ?Await, ?Return] opt DefaultClause[?Yield, ?Await, ?Return]
        CaseClauses[?Yield, ?Await, ?Return] opt }
CaseClauses[Yield, Await, Return] :
    CaseClause[?Yield, ?Await, ?Return]
    CaseClauses[?Yield, ?Await, ?Return] CaseClause[?Yield, ?Await, ?Return]
CaseClause[Yield, Await, Return] :

```

```

    case Expression [+In, ?Yield, ?Await] : StatementList [?Yield, ?Await, ?Return] opt
DefaultClause [Yield, Await, Return] :
    default : StatementList [?Yield, ?Await, ?Return] opt
LabelledStatement [Yield, Await, Return] :
    LabelIdentifier [?Yield, ?Await] : LabelledItem [?Yield, ?Await, ?Return]
LabelledItem [Yield, Await, Return] :
    Statement [?Yield, ?Await, ?Return]
    FunctionDeclaration [?Yield, ?Await, ~Default]
ThrowStatement [Yield, Await] :
    throw [no LineTerminator here] Expression [+In, ?Yield, ?Await] ;
TryStatement [Yield, Await, Return] :
    try Block [?Yield, ?Await, ?Return] Catch [?Yield, ?Await, ?Return]
    try Block [?Yield, ?Await, ?Return] Finally [?Yield, ?Await, ?Return]
    try Block [?Yield, ?Await, ?Return] Catch [?Yield, ?Await, ?Return] Finally [?Yield, ?Await, ?Return]
Catch [Yield, Await, Return] :
    catch ( CatchParameter [?Yield, ?Await] ) Block [?Yield, ?Await, ?Return]
    catch Block [?Yield, ?Await, ?Return]
Finally [Yield, Await, Return] :
    finally Block [?Yield, ?Await, ?Return]
CatchParameter [Yield, Await] :
    BindingIdentifier [?Yield, ?Await]
    BindingPattern [?Yield, ?Await]
DebuggerStatement :
    debugger ;

```

A.4 Functions and Classes

```

UniqueFormalParameters [Yield, Await] :
    FormalParameters [?Yield, ?Await]
FormalParameters [Yield, Await] :
    [empty]
    FunctionRestParameter [?Yield, ?Await]
    FormalParameterList [?Yield, ?Await]
    FormalParameterList [?Yield, ?Await] ,
    FormalParameterList [?Yield, ?Await] , FunctionRestParameter [?Yield, ?Await]
FormalParameterList [Yield, Await] :
    FormalParameter [?Yield, ?Await]
    FormalParameterList [?Yield, ?Await] , FormalParameter [?Yield, ?Await]
FunctionRestParameter [Yield, Await] :
    BindingRestElement [?Yield, ?Await]
FormalParameter [Yield, Await] :

```

*BindingElement*_[?Yield, ?Await]

*FunctionDeclaration*_[Yield, Await, Default] :

```

function BindingIdentifier[?Yield, ?Await] ( FormalParameters[~Yield, ~Await] ) {
  FunctionBody[~Yield, ~Await] }
[+Default] function ( FormalParameters[~Yield, ~Await] ) { FunctionBody[~Yield, ~Await] }

```

FunctionExpression :

```

function BindingIdentifier[~Yield, ~Await] opt ( FormalParameters[~Yield, ~Await] ) {
  FunctionBody[~Yield, ~Await] }

```

*FunctionBody*_[Yield, Await] :

```

FunctionStatementList[?Yield, ?Await]

```

*FunctionStatementList*_[Yield, Await] :

```

StatementList[?Yield, ?Await, +Return] opt

```

*ArrowFunction*_[In, Yield, Await] :

```

ArrowParameters[?Yield, ?Await] [no LineTerminator here] => ConciseBody[?In]

```

*ArrowParameters*_[Yield, Await] :

```

BindingIdentifier[?Yield, ?Await]
CoverParenthesizedExpressionAndArrowParameterList[?Yield, ?Await]

```

*ConciseBody*_[In] :

```

[lookahead ≠ {] ExpressionBody[?In, ~Await]
{ FunctionBody[~Yield, ~Await] }

```

*ExpressionBody*_[In, Await] :

```

AssignmentExpression[?In, ~Yield, ?Await]

```

When processing an instance of the production

*ArrowParameters*_[Yield, Await] : *CoverParenthesizedExpressionAndArrowParameterList*_[?Yield, ?Await]

the interpretation of *CoverParenthesizedExpressionAndArrowParameterList* is refined using the following grammar:

*ArrowFormalParameters*_[Yield, Await] :

```

( UniqueFormalParameters[?Yield, ?Await] )

```

*AsyncArrowFunction*_[In, Yield, Await] :

```

async [no LineTerminator here] AsyncArrowBindingIdentifier[?Yield] [no LineTerminator here] =>
  AsyncConciseBody[?In]
CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await] [no LineTerminator here] => AsyncConciseBody[?In]

```

*AsyncConciseBody*_[In] :

```

[lookahead ≠ {] ExpressionBody[?In, +Await]
{ AsyncFunctionBody }

```

*AsyncArrowBindingIdentifier*_[Yield] :

```

BindingIdentifier[?Yield, +Await]

```

*CoverCallExpressionAndAsyncArrowHead*_[Yield, Await] :

```

MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]

```

When processing an instance of the production

*AsyncArrowFunction*_[In, Yield, Await] : *CoverCallExpressionAndAsyncArrowHead*_[?Yield, ?Await] [no *LineTerminator* here] => *AsyncConciseBody*_[?In]

the interpretation of *CoverCallExpressionAndAsyncArrowHead* is refined using the following grammar:

AsyncArrowHead :

async [no *LineTerminator* here] *ArrowFormalParameters*_[~Yield, +Await]

*MethodDefinition*_[Yield, Await] :

*PropertyName*_[?Yield, ?Await] (*UniqueFormalParameters*_[~Yield, ~Await]) { *FunctionBody*_[~Yield, ~Await] }

*GeneratorMethod*_[?Yield, ?Await]

*AsyncMethod*_[?Yield, ?Await]

*AsyncGeneratorMethod*_[?Yield, ?Await]

get *PropertyName*_[?Yield, ?Await] () { *FunctionBody*_[~Yield, ~Await] }

set *PropertyName*_[?Yield, ?Await] (*PropertySetParameterList*) { *FunctionBody*_[~Yield, ~Await] }

PropertySetParameterList :

*FormalParameter*_[~Yield, ~Await]

*GeneratorMethod*_[Yield, Await] :

* *PropertyName*_[?Yield, ?Await] (*UniqueFormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

*GeneratorDeclaration*_[Yield, Await, Default] :

function * *BindingIdentifier*_[?Yield, ?Await] (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

[+Default] **function** * (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

GeneratorExpression :

function * *BindingIdentifier*_[+Yield, ~Await] *opt* (*FormalParameters*_[+Yield, ~Await]) { *GeneratorBody* }

GeneratorBody :

*FunctionBody*_[+Yield, ~Await]

*YieldExpression*_[In, Await] :

yield

yield [no *LineTerminator* here] *AssignmentExpression*_[?In, +Yield, ?Await]

yield [no *LineTerminator* here] * *AssignmentExpression*_[?In, +Yield, ?Await]

*AsyncGeneratorMethod*_[Yield, Await] :

async [no *LineTerminator* here] * *PropertyName*_[?Yield, ?Await] (*UniqueFormalParameters*_[+Yield, +Await])
{ *AsyncGeneratorBody* }

*AsyncGeneratorDeclaration*_[Yield, Await, Default] :

async [no *LineTerminator* here] **function** * *BindingIdentifier*_[?Yield, ?Await] (
*FormalParameters*_[+Yield, +Await]) { *AsyncGeneratorBody* }

[+Default] **async** [no *LineTerminator* here] **function** * (*FormalParameters*_[+Yield, +Await]) {
AsyncGeneratorBody }

AsyncGeneratorExpression :

async [no *LineTerminator* here] **function** * *BindingIdentifier*_[+Yield, +Await] *opt* (
*FormalParameters*_[+Yield, +Await]) { *AsyncGeneratorBody* }

AsyncGeneratorBody :

```

    FunctionBody [+Yield, +Await]
AsyncFunctionDeclaration [Yield, Await, Default] :
    async [no LineTerminator here] function BindingIdentifier [?Yield, ?Await] ( FormalParameters [~Yield, +Await]
        ) { AsyncFunctionBody }
    [+Default] async [no LineTerminator here] function ( FormalParameters [~Yield, +Await] ) { AsyncFunctionBody
        }
AsyncFunctionExpression :
    async [no LineTerminator here] function BindingIdentifier [~Yield, +Await] opt (
        FormalParameters [~Yield, +Await] ) { AsyncFunctionBody }
AsyncMethod [Yield, Await] :
    async [no LineTerminator here] PropertyName [?Yield, ?Await] ( UniqueFormalParameters [~Yield, +Await] ) {
        AsyncFunctionBody }
AsyncFunctionBody :
    FunctionBody [~Yield, +Await]
AwaitExpression [Yield] :
    await UnaryExpression [?Yield, +Await]
ClassDeclaration [Yield, Await, Default] :
    class BindingIdentifier [?Yield, ?Await] ClassTail [?Yield, ?Await]
    [+Default] class ClassTail [?Yield, ?Await]
ClassExpression [Yield, Await] :
    class BindingIdentifier [?Yield, ?Await] opt ClassTail [?Yield, ?Await]
ClassTail [Yield, Await] :
    ClassHeritage [?Yield, ?Await] opt { ClassBody [?Yield, ?Await] opt }
ClassHeritage [Yield, Await] :
    extends LeftHandSideExpression [?Yield, ?Await]
ClassBody [Yield, Await] :
    ClassElementList [?Yield, ?Await]
ClassElementList [Yield, Await] :
    ClassElement [?Yield, ?Await]
    ClassElementList [?Yield, ?Await] ClassElement [?Yield, ?Await]
ClassElement [Yield, Await] :
    MethodDefinition [?Yield, ?Await]
    static MethodDefinition [?Yield, ?Await]
    ;

```

A.5 Scripts and Modules

```

Script :
    ScriptBody opt
ScriptBody :
    StatementList [~Yield, ~Await, ~Return]
Module :

```

```

    ModuleBodyopt
ModuleBody :
    ModuleItemList
ModuleItemList :
    ModuleItem
    ModuleItemList ModuleItem
ModuleItem :
    ImportDeclaration
    ExportDeclaration
    StatementListItem [~Yield, ~Await, ~Return]
ImportDeclaration :
    import ImportClause FromClause ;
    import ModuleSpecifier ;
ImportClause :
    ImportedDefaultBinding
    NamespaceImport
    NamedImports
    ImportedDefaultBinding , NamespaceImport
    ImportedDefaultBinding , NamedImports
ImportedDefaultBinding :
    ImportedBinding
NamespaceImport :
    * as ImportedBinding
NamedImports :
    { }
    { ImportsList }
    { ImportsList , }
FromClause :
    from ModuleSpecifier
ImportsList :
    ImportSpecifier
    ImportsList , ImportSpecifier
ImportSpecifier :
    ImportedBinding
    IdentifierName as ImportedBinding
ModuleSpecifier :
    StringLiteral
ImportedBinding :
    BindingIdentifier [~Yield, ~Await]
ExportDeclaration :
    export ExportFromClause FromClause ;
    export NamedExports ;
    export VariableStatement [~Yield, ~Await]
    export Declaration [~Yield, ~Await]

```

```

export default HoistableDeclaration [~Yield, ~Await, +Default]
export default ClassDeclaration [~Yield, ~Await, +Default]
export default [lookahead ∉ { function, async [no LineTerminator here] function, class }]
  AssignmentExpression [+In, ~Yield, ~Await] ;

```

ExportFromClause :

```

*
* as IdentifierName
NamedExports

```

NamedExports :

```

{ }
{ ExportsList }
{ ExportsList , }

```

ExportsList :

```

ExportSpecifier
ExportsList , ExportSpecifier

```

ExportSpecifier :

```

IdentifierName
IdentifierName as IdentifierName

```

A.6 Number Conversions

StringNumericLiteral :::

```

StrWhiteSpaceopt
StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt

```

StrWhiteSpace :::

```

StrWhiteSpaceChar StrWhiteSpaceopt

```

StrWhiteSpaceChar :::

```

WhiteSpace
LineTerminator

```

StrNumericLiteral :::

```

StrDecimalLiteral
NonDecimalIntegerLiteral [~Sep]

```

StrDecimalLiteral :::

```

StrUnsignedDecimalLiteral
+ StrUnsignedDecimalLiteral
- StrUnsignedDecimalLiteral

```

StrUnsignedDecimalLiteral :::

```

Infinity
DecimalDigits [~Sep] . DecimalDigits [~Sep] opt ExponentPart [~Sep] opt
. DecimalDigits [~Sep] ExponentPart [~Sep] opt
DecimalDigits [~Sep] ExponentPart [~Sep] opt

```

All grammar symbols not explicitly defined by the *StringNumericLiteral* grammar have the definitions used in the [Lexical Grammar for numeric literals](#).

A.7 Universal Resource Identifier Character Classes

uri ::=
 *uriCharacters*_{opt}

uriCharacters ::=
 uriCharacter *uriCharacters*_{opt}

uriCharacter ::=
 uriReserved
 uriUnescaped
 uriEscaped

uriReserved ::= **one of**
 ; / ? : @ & = + \$,

uriUnescaped ::=
 uriAlpha
 DecimalDigit
 uriMark

uriEscaped ::=
 % *HexDigit* *HexDigit*

uriAlpha ::= **one of**
 a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

uriMark ::= **one of**
 - _ . ! ~ * ' ()

A.8 Regular Expressions

*Pattern*_[U, N] ::=
 *Disjunction*_[?U, ?N]

*Disjunction*_[U, N] ::=
 *Alternative*_[?U, ?N]
 *Alternative*_[?U, ?N] | *Disjunction*_[?U, ?N]

*Alternative*_[U, N] ::=
 [empty]
 *Alternative*_[?U, ?N] *Term*_[?U, ?N]

*Term*_[U, N] ::=
 *Assertion*_[?U, ?N]
 *Atom*_[?U, ?N]
 *Atom*_[?U, ?N] *Quantifier*

*Assertion*_[U, N] ::=
 ^
 \$
 \ b
 \ B
 (? = *Disjunction*_[?U, ?N])
 (? ! *Disjunction*_[?U, ?N])

(? <= Disjunction [?U, ?N])

(? <! Disjunction [?U, ?N])

Quantifier ::

QuantifierPrefix

QuantifierPrefix ?

QuantifierPrefix ::

*

+

?

{ DecimalDigits [~Sep] }

{ DecimalDigits [~Sep] , }

{ DecimalDigits [~Sep] , DecimalDigits [~Sep] }

Atom [U, N] ::

PatternCharacter

.

\ AtomEscape [?U, ?N]

CharacterClass [?U]

(GroupSpecifier [?U] Disjunction [?U, ?N])

(? : Disjunction [?U, ?N])

SyntaxCharacter :: **one of**

^ \$ \ . * + ? () [] { } |

PatternCharacter ::

SourceCharacter but not SyntaxCharacter

AtomEscape [U, N] ::

DecimalEscape

CharacterClassEscape [?U]

CharacterEscape [?U]

[+N] k GroupName [?U]

CharacterEscape [U] ::

ControlEscape

c ControlLetter

0 [lookahead \notin DecimalDigit]

HexEscapeSequence

RegExpUnicodeEscapeSequence [?U]

IdentityEscape [?U]

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

GroupSpecifier [U] ::

[empty]

? GroupName [?U]

*GroupName*_[U] ::
< *RegExpIdentifierName*_[?U] >

*RegExpIdentifierName*_[U] ::
*RegExpIdentifierStart*_[?U]
*RegExpIdentifierName*_[?U] *RegExpIdentifierPart*_[?U]

*RegExpIdentifierStart*_[U] ::
UnicodeIDStart
\$
-
\ *RegExpUnicodeEscapeSequence*_[+U]
[~U] *UnicodeLeadSurrogate* *UnicodeTrailSurrogate*

*RegExpIdentifierPart*_[U] ::
UnicodeIDContinue
\$
\ *RegExpUnicodeEscapeSequence*_[+U]
[~U] *UnicodeLeadSurrogate* *UnicodeTrailSurrogate*
<ZWNJ>
<ZWJ>

*RegExpUnicodeEscapeSequence*_[U] ::
[+U] **u** *HexLeadSurrogate* \ **u** *HexTrailSurrogate*
[+U] **u** *HexLeadSurrogate*
[+U] **u** *HexTrailSurrogate*
[+U] **u** *HexNonSurrogate*
[~U] **u** *Hex4Digits*
[+U] **u**{ *CodePoint* }

UnicodeLeadSurrogate ::
any Unicode code point in the inclusive range 0xD800 to 0xDBFF

UnicodeTrailSurrogate ::
any Unicode code point in the inclusive range 0xDC00 to 0xDFFF

Each \ **u** *HexTrailSurrogate* for which the choice of associated **u** *HexLeadSurrogate* is ambiguous shall be associated with the nearest possible **u** *HexLeadSurrogate* that would otherwise have no corresponding \ **u** *HexTrailSurrogate*.

HexLeadSurrogate ::
Hex4Digits but only if the MV of *Hex4Digits* is in the inclusive range 0xD800 to 0xDBFF

HexTrailSurrogate ::
Hex4Digits but only if the MV of *Hex4Digits* is in the inclusive range 0xDC00 to 0xDFFF

HexNonSurrogate ::
Hex4Digits but only if the MV of *Hex4Digits* is not in the inclusive range 0xD800 to 0xDFFF

*IdentityEscape*_[U] ::
[+U] *SyntaxCharacter*
[+U] /
[~U] *SourceCharacter* but not *UnicodeIDContinue*

DecimalEscape ::
NonZeroDigit *DecimalDigits* [*~Sep*] *opt* [*lookahead* ∉ *DecimalDigit*]

CharacterClassEscape [*U*] ::
d
D
s
S
w
W
[+*U*] **p**{ *UnicodePropertyValueExpression* }
[+*U*] **P**{ *UnicodePropertyValueExpression* }

UnicodePropertyValueExpression ::
UnicodePropertyName = *UnicodePropertyValue*
LoneUnicodePropertyNameOrValue

UnicodePropertyName ::
UnicodePropertyNameCharacters

UnicodePropertyNameCharacters ::
UnicodePropertyNameCharacter *UnicodePropertyNameCharacters*_{*opt*}

UnicodePropertyValue ::
UnicodePropertyValueCharacters

LoneUnicodePropertyNameOrValue ::
UnicodePropertyValueCharacters

UnicodePropertyValueCharacters ::
UnicodePropertyValueCharacter *UnicodePropertyValueCharacters*_{*opt*}

UnicodePropertyValueCharacter ::
UnicodePropertyNameCharacter
DecimalDigit

UnicodePropertyNameCharacter ::
ControlLetter
–

CharacterClass [*U*] ::
[[*lookahead* ≠ ^] *ClassRanges* [*?U*]]
[^ *ClassRanges* [*?U*]]

ClassRanges [*U*] ::
[empty]
NonemptyClassRanges [*?U*]

NonemptyClassRanges [*U*] ::
ClassAtom [*?U*]
ClassAtom [*?U*] *NonemptyClassRangesNoDash* [*?U*]
ClassAtom [*?U*] – *ClassAtom* [*?U*] *ClassRanges* [*?U*]

NonemptyClassRangesNoDash [*U*] ::
ClassAtom [*?U*]
ClassAtomNoDash [*?U*] *NonemptyClassRangesNoDash* [*?U*]

*ClassAtomNoDash*_[?U] - *ClassAtom*_[?U] *ClassRanges*_[?U]

*ClassAtom*_[U] ::

-

*ClassAtomNoDash*_[?U]

*ClassAtomNoDash*_[U] ::

SourceCharacter but not one of \ or] or -

\ *ClassEscape*_[?U]

*ClassEscape*_[U] ::

b

[+U] -

*CharacterClassEscape*_[?U]

*CharacterEscape*_[?U]

B 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.

NOTE

This annex describes various legacy features and other characteristics of web browser ECMAScript hosts. All of the language features and behaviours specified in this annex have one or more undesirable characteristics and in the absence of legacy usage would be removed from this specification. However, the usage of these features by large numbers of existing web pages means that web browsers must continue to support them. The specifications in this annex define the requirements for interoperable implementations of these legacy features.

These features are not considered part of the core ECMAScript language. Programmers should not use or assume the existence of these features and behaviours when writing new ECMAScript code. ECMAScript implementations are discouraged from implementing these features unless the implementation is part of a web browser or is required to run the same legacy ECMAScript code that web browsers encounter.

B.1 Additional Syntax

B.1.1 Numeric Literals

The syntax and semantics of [12.8.3](#) is extended as follows except that this extension is not allowed for [strict mode code](#):

Syntax

NumericLiteral ::

DecimalLiteral

DecimalBigIntegerLiteral

NonDecimalIntegerLiteral ^[+Sep]
NonDecimalIntegerLiteral ^[+Sep] *BigIntLiteralSuffix*
LegacyOctalIntegerLiteral

LegacyOctalIntegerLiteral ::
 0 *OctalDigit*
 LegacyOctalIntegerLiteral *OctalDigit*

DecimalIntegerLiteral ::
 0
 NonZeroDigit
 NonZeroDigit *NumericLiteralSeparator*_{opt} *DecimalDigits*^[+Sep]
 NonOctalDecimalIntegerLiteral

NonOctalDecimalIntegerLiteral ::
 0 *NonOctalDigit*
 LegacyOctalLikeDecimalIntegerLiteral *NonOctalDigit*
 NonOctalDecimalIntegerLiteral *DecimalDigit*

LegacyOctalLikeDecimalIntegerLiteral ::
 0 *OctalDigit*
 LegacyOctalLikeDecimalIntegerLiteral *OctalDigit*

NonOctalDigit :: **one of**
 8 9

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*.
- The MV of *DecimalIntegerLiteral* :: *NonOctalDecimalIntegerLiteral* is the MV of *NonOctalDecimalIntegerLiteral*.
- The MV of *NonOctalDecimalIntegerLiteral* :: **0** *NonOctalDigit* is the MV of *NonOctalDigit*.
- The MV of *NonOctalDecimalIntegerLiteral* :: *LegacyOctalLikeDecimalIntegerLiteral* *NonOctalDigit* is (the MV of *LegacyOctalLikeDecimalIntegerLiteral* times 10) plus the MV of *NonOctalDigit*.
- The MV of *NonOctalDecimalIntegerLiteral* :: *NonOctalDecimalIntegerLiteral* *DecimalDigit* is (the MV of *NonOctalDecimalIntegerLiteral* times 10) plus the MV of *DecimalDigit*.
- The MV of *LegacyOctalLikeDecimalIntegerLiteral* :: **0** *OctalDigit* is the MV of *OctalDigit*.
- The MV of *LegacyOctalLikeDecimalIntegerLiteral* :: *LegacyOctalLikeDecimalIntegerLiteral* *OctalDigit* is (the MV of *LegacyOctalLikeDecimalIntegerLiteral* times 10) plus the MV of *OctalDigit*.
- The MV of *NonOctalDigit* :: **8** is 8.
- The MV of *NonOctalDigit* :: **9** is 9.

B.1.2 String Literals

The syntax and semantics of 12.8.4 is extended as follows except that this extension is not allowed for [strict mode code](#):

Syntax

EscapeSequence ::

CharacterEscapeSequence

LegacyOctalEscapeSequence

NonOctalDecimalEscapeSequence

HexEscapeSequence

UnicodeEscapeSequence

LegacyOctalEscapeSequence ::

OctalDigit [lookahead \notin *OctalDigit*]

ZeroToThree *OctalDigit* [lookahead \notin *OctalDigit*]

FourToSeven *OctalDigit*

ZeroToThree *OctalDigit* *OctalDigit*

ZeroToThree :: **one of**

0 1 2 3

FourToSeven :: **one of**

4 5 6 7

NonOctalDecimalEscapeSequence :: **one of**

8 9

This definition of *EscapeSequence* is not used in strict mode or when parsing *TemplateCharacter*.

NOTE

It is possible for string literals to precede a [Use Strict Directive](#) that places the enclosing code in [strict mode](#), and implementations must take care to not use this extended definition of *EscapeSequence* with such literals. For example, attempting to parse the following source text must fail:

```
function invalid() { "\7"; "use strict"; }
```

B.1.2.1 Static Semantics

- The **SV** of *EscapeSequence* :: *LegacyOctalEscapeSequence* is the String value consisting of the code unit whose value is the MV of *LegacyOctalEscapeSequence*.
- The MV of *LegacyOctalEscapeSequence* :: *ZeroToThree* *OctalDigit* is (8 times the MV of *ZeroToThree*) plus the MV of *OctalDigit*.
- The MV of *LegacyOctalEscapeSequence* :: *FourToSeven* *OctalDigit* is (8 times the MV of *FourToSeven*) plus the MV of *OctalDigit*.
- The MV of *LegacyOctalEscapeSequence* :: *ZeroToThree* *OctalDigit* *OctalDigit* is (64 (that is, 8^2) times the MV of *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The **SV** of *NonOctalDecimalEscapeSequence* :: **8** is the String value consisting of the code unit 0x0038 (DIGIT EIGHT).
- The **SV** of *NonOctalDecimalEscapeSequence* :: **9** is the String value consisting of the code unit 0x0039 (DIGIT NINE).

- 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

The syntax and semantics of 12.4 is extended as follows except that this extension is not allowed when parsing source code using the *goal symbol Module*:

Syntax

Comment ::

MultiLineComment

SingleLineComment

SingleLineHTMLOpenComment

SingleLineHTMLCloseComment

SingleLineDelimitedComment

MultiLineComment ::

/ FirstCommentLine_{opt} LineTerminator MultiLineCommentChars_{opt} */ HTMLCloseComment_{opt}*

FirstCommentLine ::

SingleLineDelimitedCommentChars

SingleLineHTMLOpenComment ::

<!-- SingleLineCommentChars_{opt}

SingleLineHTMLCloseComment ::

LineTerminatorSequence HTMLCloseComment

SingleLineDelimitedComment ::

/ SingleLineDelimitedCommentChars_{opt} */*

HTMLCloseComment ::

WhiteSpaceSequence_{opt} SingleLineDelimitedCommentSequence_{opt} --> SingleLineCommentChars_{opt}

SingleLineDelimitedCommentChars ::

SingleLineNotAsteriskChar SingleLineDelimitedCommentChars_{opt}

** SingleLinePostAsteriskCommentChars_{opt}*

SingleLineNotAsteriskChar ::

SourceCharacter but not one of *** or *LineTerminator*

SingleLinePostAsteriskCommentChars ::

SingleLineNotForwardSlashOrAsteriskChar *SingleLineDelimitedCommentChars*_{opt}

* *SingleLinePostAsteriskCommentChars*_{opt}

SingleLineNotForwardSlashOrAsteriskChar ::

SourceCharacter but not one of / or * or *LineTerminator*

WhiteSpaceSequence ::

WhiteSpace *WhiteSpaceSequence*_{opt}

SingleLineDelimitedCommentSequence ::

SingleLineDelimitedComment *WhiteSpaceSequence*_{opt} *SingleLineDelimitedCommentSequence*_{opt}

Similar to a *MultiLineComment* that contains a line terminator code point, a *SingleLineHTMLCloseComment* is considered to be a *LineTerminator* for purposes of parsing by the syntactic grammar.

B.1.4 Regular Expressions Patterns

The syntax of 22.2.1 is modified and extended as follows. These changes introduce ambiguities that are broken by the ordering of grammar productions and by contextual information. When parsing using the following grammar, each alternative is considered only if previous production alternatives do not match.

This alternative pattern grammar and semantics only changes the syntax and semantics of BMP patterns. The following grammar extensions include productions parameterized with the [U] parameter. However, none of these extensions change the syntax of Unicode patterns recognized when parsing with the [U] parameter present on the [goal symbol](#).

Syntax

*Term*_[U, N] ::

[+U] *Assertion*_[+U, ?N]

[+U] *Atom*_[+U, ?N] *Quantifier*

[+U] *Atom*_[+U, ?N]

[~U] *QuantifiableAssertion*_[?N] *Quantifier*

[~U] *Assertion*_[~U, ?N]

[~U] *ExtendedAtom*_[?N] *Quantifier*

[~U] *ExtendedAtom*_[?N]

*Assertion*_[U, N] ::

^

\$

\ b

\ B

[+U] (? = *Disjunction*_[+U, ?N])

[+U] (? ! *Disjunction*_[+U, ?N])

[~U] *QuantifiableAssertion*_[?N]

(? <= *Disjunction*_[?U, ?N])

(? <! Disjunction [?U, ?N])

QuantifiableAssertion [N] ::

(? = Disjunction [~U, ?N])

(? ! Disjunction [~U, ?N])

ExtendedAtom [N] ::

.

\ AtomEscape [~U, ?N]

\ [lookahead = c]

CharacterClass [~U]

(Disjunction [~U, ?N])

(? : Disjunction [~U, ?N])

InvalidBracedQuantifier

ExtendedPatternCharacter

InvalidBracedQuantifier ::

{ DecimalDigits [~Sep] }

{ DecimalDigits [~Sep] , }

{ DecimalDigits [~Sep] , DecimalDigits [~Sep] }

ExtendedPatternCharacter ::

SourceCharacter but not one of ^ \$ \ . * + ? () [|

AtomEscape [U, N] ::

[+U] DecimalEscape

[~U] DecimalEscape but only if the CapturingGroupNumber of DecimalEscape is ≤ NcapturingParens

CharacterClassEscape [?U]

CharacterEscape [?U, ?N]

[+N] k GroupName [?U]

CharacterEscape [U, N] ::

ControlEscape

c ControlLetter

∅ [lookahead ∉ DecimalDigit]

HexEscapeSequence

RegExpUnicodeEscapeSequence [?U]

[~U] LegacyOctalEscapeSequence

IdentityEscape [?U, ?N]

IdentityEscape [U, N] ::

[+U] SyntaxCharacter

[+U] /

[~U] SourceCharacterIdentityEscape [?N]

*SourceCharacterIdentityEscape*_[N] ::
 [-N] *SourceCharacter* but not **c**
 [+N] *SourceCharacter* but not one of **c** or **k**

*ClassAtomNoDash*_[U, N] ::
SourceCharacter but not one of \ or] or -
 \ *ClassEscape*_[?U, ?N]
 \ [lookahead = **c**]

*ClassEscape*_[U, N] ::
b
 [+U] -
 [-U] **c** *ClassControlLetter*
*CharacterClassEscape*_[?U]
*CharacterEscape*_[?U, ?N]

ClassControlLetter ::
DecimalDigit
 -

NOTE When the same left hand sides occurs with both [+U] and [-U] guards it is to control the disambiguation priority.

B.1.4.1 Static Semantics: Early Errors

The semantics of 22.2.1.1 is extended as follows:

ExtendedAtom :: *InvalidBracedQuantifier*

- It is a Syntax Error if any source text matches this rule.

Additionally, the rules for the following productions are modified with the addition of the highlighted text:

NonemptyClassRanges :: *ClassAtom* - *ClassAtom* *ClassRanges*

- It is a Syntax Error if *IsCharacterClass* of the first *ClassAtom* is **true** or *IsCharacterClass* of the second *ClassAtom* is **true** and this production has a [U] parameter.
- It is a Syntax Error if *IsCharacterClass* of the first *ClassAtom* is **false** and *IsCharacterClass* of the second *ClassAtom* is **false** and the *CharacterValue* of the first *ClassAtom* is larger than the *CharacterValue* of the second *ClassAtom*.

NonemptyClassRangesNoDash :: *ClassAtomNoDash* - *ClassAtom* *ClassRanges*

- It is a Syntax Error if *IsCharacterClass* of *ClassAtomNoDash* is **true** or *IsCharacterClass* of *ClassAtom* is **true** and this production has a [U] parameter.
- It is a Syntax Error if *IsCharacterClass* of *ClassAtomNoDash* is **false** and *IsCharacterClass* of *ClassAtom* is **false** and the *CharacterValue* of *ClassAtomNoDash* is larger than the *CharacterValue* of *ClassAtom*.

B.1.4.2 Static Semantics: IsCharacterClass

The semantics of 22.2.1.3 is extended as follows:

ClassAtomNoDash :: \ [lookahead = c]

1. Return **false**.

B.1.4.3 Static Semantics: CharacterValue

The semantics of 22.2.1.4 is extended as follows:

ClassAtomNoDash :: \ [lookahead = c]

1. Return the code point value of U+005C (REVERSE SOLIDUS).

ClassEscape :: c *ClassControlLetter*

1. Let *ch* be the code point matched by *ClassControlLetter*.
2. Let *i* be *ch*'s code point value.
3. Return the remainder of dividing *i* by 32.

CharacterEscape :: *LegacyOctalEscapeSequence*

1. Return the MV of *LegacyOctalEscapeSequence* (see B.1.2).

B.1.4.4 Pattern Semantics

The semantics of 22.2.2 is extended as follows:

Within 22.2.2.5 reference to “ *Atom* :: (*GroupSpecifier Disjunction*) ” are to be interpreted as meaning “ *Atom* :: (*GroupSpecifier Disjunction*) ” or “ *ExtendedAtom* :: (*Disjunction*) ”.

Term (22.2.2.5) includes the following additional evaluation rules:

The production *Term* :: *QuantifiableAssertion Quantifier* evaluates the same as the production *Term* :: *Atom Quantifier* but with *QuantifiableAssertion* substituted for *Atom*.

The production *Term* :: *ExtendedAtom Quantifier* evaluates the same as the production *Term* :: *Atom Quantifier* but with *ExtendedAtom* substituted for *Atom*.

The production *Term* :: *ExtendedAtom* evaluates the same as the production *Term* :: *Atom* but with *ExtendedAtom* substituted for *Atom*.

Assertion (22.2.2.6) includes the following additional evaluation rule:

The production *Assertion* :: *QuantifiableAssertion* evaluates as follows:

1. Evaluate *QuantifiableAssertion* to obtain a Matcher *m*.
2. Return *m*.

Assertion (22.2.2.6) evaluation rules for the *Assertion* :: (? = *Disjunction*) and *Assertion* :: (? ! *Disjunction*) productions are also used for the *QuantifiableAssertion* productions, but with *QuantifiableAssertion* substituted for *Assertion*.

Atom (22.2.2.8) evaluation rules for the *Atom* productions except for *Atom* :: *PatternCharacter* are also used for the *ExtendedAtom* productions, but with *ExtendedAtom* substituted for *Atom*. The following evaluation rules, with parameter *direction*, are also added:

The production *ExtendedAtom* :: \ [lookahead = c] evaluates as follows:

1. Let *A* be the CharSet containing the single character \ U+005C (REVERSE SOLIDUS).
2. Return ! **CharacterSetMatcher**(*A*, **false**, *direction*).

The production *ExtendedAtom* :: *ExtendedPatternCharacter* evaluates as follows:

1. Let *ch* be the character represented by *ExtendedPatternCharacter*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Return ! **CharacterSetMatcher**(*A*, **false**, *direction*).

CharacterEscape (22.2.2.10) includes the following additional evaluation rule:

The production *CharacterEscape* :: *LegacyOctalEscapeSequence* evaluates as follows:

1. Let *cv* be the **CharacterValue** of this *CharacterEscape*.
2. Return the character whose character value is *cv*.

NonemptyClassRanges (22.2.2.15) modifies the following evaluation rule:

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. Let *D* be ! **CharacterRangeOrUnion**(*A*, *B*).
5. Return the union of *D* and *C*.

NonemptyClassRangesNoDash (22.2.2.16) modifies the following evaluation rule:

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. Let *D* be ! **CharacterRangeOrUnion**(*A*, *B*).
5. Return the union of *D* and *C*.

ClassEscape (22.2.2.19) includes the following additional evaluation rule:

The production *ClassEscape* :: c *ClassControlLetter* evaluates as follows:

1. Let *cv* be the **CharacterValue** of this *ClassEscape*.
2. Let *c* be the character whose character value is *cv*.
3. Return the CharSet containing the single character *c*.

ClassAtomNoDash (22.2.2.18) includes the following additional evaluation rule:

The production *ClassAtomNoDash* :: \ [lookahead = c] evaluates as follows:

1. Return the CharSet containing the single character `\ U+005C` (REVERSE SOLIDUS).

NOTE

This production can only be reached from the sequence `\c` within a character class where it is not followed by an acceptable control character.

B.1.4.4.1 CharacterRangeOrUnion (*A*, *B*)

The abstract operation `CharacterRangeOrUnion` takes arguments *A* (a CharSet) and *B* (a CharSet). It performs the following steps when called:

1. If *Unicode* is **false**, then
 - a. If *A* does not contain exactly one character or *B* does not contain exactly one character, then
 - i. Let *C* be the CharSet containing the single character `- U+002D` (HYPHEN-MINUS).
 - ii. Return the union of CharSets *A*, *B* and *C*.
2. Return ! `CharacterRange(A, B)`.

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

The entries in [Table 82](#) are added to [Table 8](#).

Table 82: Additional Well-known Intrinsic Objects

Intrinsic Name	Global Name	ECMAScript Language Association
<code>%escape%</code>	escape	The escape function (B.2.1.1)
<code>%unescape%</code>	unescape	The unescape function (B.2.1.2)

B.2.1.1 escape (*string*)

The **escape** function is a property of the [global object](#). It computes a new version of a String value in which certain code units have been replaced by a hexadecimal escape sequence.

For those code units being replaced whose value is `0x00FF` 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 `0x00FF`, a four-digit escape sequence of the form `%uxxxx` is used.

The **escape** function is the `%escape%` intrinsic object. When the **escape** function is called with one argument *string*, the following steps are taken:

1. Set *string* to ? `ToString(string)`.
2. Let *length* be the number of code units in *string*.
3. Let *R* be the empty String.

4. Let k be 0.
5. Repeat, while $k < \text{length}$,
 - a. Let char be the code unit (represented as a 16-bit unsigned **integer**) at index k within string .
 - b. If char is one of the code units in "ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-.!", then
 - i. Let S be the String value containing the single code unit char .
 - c. Else if $\text{char} \geq 256$, then
 - i. Let n be the numeric value of char .
 - ii. Let S be the **string-concatenation** of:
 - "%u"
 - the String representation of n , formatted as a four-digit uppercase hexadecimal number, padded to the left with zeroes if necessary
 - d. Else,
 - i. **Assert:** $\text{char} < 256$.
 - ii. Let n be the numeric value of char .
 - iii. Let S be the **string-concatenation** of:
 - "%"
 - the String representation of n , formatted as a two-digit uppercase hexadecimal number, padded to the left with a zero if necessary
 - e. Set R to the **string-concatenation** of R and S .
 - f. Set k to $k + 1$.
6. Return R .

NOTE

The encoding is partly based on the encoding described in RFC 1738, but the entire encoding specified in this standard is described above without regard to the contents of RFC 1738. This encoding does not reflect changes to RFC 1738 made by RFC 3986.

B.2.1.2 unescape (*string*)

The **unescape** function is a property of the **global object**. It computes a new version of a String value in which each escape sequence of the sort that might be introduced by the **escape** function is replaced with the code unit that it represents.

The **unescape** function is the `%unescape%` intrinsic object. When the **unescape** function is called with one argument *string*, the following steps are taken:

1. Set *string* to ? **ToString**(*string*).
2. Let *length* be the number of code units in *string*.
3. Let R be the empty String.
4. Let k be 0.
5. Repeat, while $k \neq \text{length}$,
 - a. Let c be the code unit at index k within *string*.
 - b. If c is the code unit 0x0025 (PERCENT SIGN), then
 - i. Let *hexEscape* be the empty String.
 - ii. Let *skip* be 0.
 - iii. If $k \leq \text{length} - 6$ and the code unit at index $k + 1$ within *string* is the code unit 0x0075 (LATIN SMALL LETTER U), then
 1. Set *hexEscape* to the **substring** of *string* from $k + 2$ to $k + 6$.

2. Set *skip* to 5.
 - iv. Else if $k \leq \text{length} - 3$, then
 1. Set *hexEscape* to the **substring** of *string* from $k + 1$ to $k + 3$.
 2. Set *skip* to 2.
 - v. If *hexEscape* can be interpreted as an expansion of *HexDigits*_[~Sep], then
 1. Let *hexIntegerLiteral* be the **string-concatenation** of "0x" and *hexEscape*.
 2. Let *n* be ! **ToNumber**(*hexIntegerLiteral*).
 3. Set *c* to the code unit whose value is $\mathbb{R}(n)$.
 4. Set *k* to $k + \text{skip}$.
 - c. Set *R* to the **string-concatenation** of *R* and *c*.
 - d. Set *k* to $k + 1$.
6. 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 when called:

1. Let *O* be ? **ToObject**(**this** value).
2. Return ? *O*.**[[GetPrototypeOf]]**() .

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 when called:

1. Let *O* be ? **RequireObjectCoercible**(**this** value).
2. If **Type**(*proto*) is neither Object nor Null, return **undefined**.
3. If **Type**(*O*) is not Object, return **undefined**.
4. Let *status* be ? *O*.**[[SetPrototypeOf]]**(*proto*).
5. If *status* is **false**, throw a **TypeError** exception.
6. Return **undefined**.

B.2.2.2 Object.prototype.__defineGetter__ (*P*, *getter*)

When the **__defineGetter__** method is called with arguments *P* and *getter*, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. If **IsCallable**(*getter*) is **false**, throw a **TypeError** exception.
3. Let *desc* be PropertyDescriptor { **[[Get]]**: *getter*, **[[Enumerable]]**: **true**, **[[Configurable]]**: **true** }.
4. Let *key* be ? **ToPropertyKey**(*P*).
5. Perform ? **DefinePropertyOrThrow**(*O*, *key*, *desc*).

6. Return **undefined**.

B.2.2.3 Object.prototype.__defineSetter__ (*P*, *setter*)

When the **__defineSetter__** method is called with arguments *P* and *setter*, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. If **IsCallable**(*setter*) is **false**, throw a **TypeError** exception.
3. Let *desc* be PropertyDescriptor { **[[Set]]**: *setter*, **[[Enumerable]]**: **true**, **[[Configurable]]**: **true** }.
4. Let *key* be ? **ToPropertyKey**(*P*).
5. Perform ? **DefinePropertyOrThrow**(*O*, *key*, *desc*).
6. Return **undefined**.

B.2.2.4 Object.prototype.__lookupGetter__ (*P*)

When the **__lookupGetter__** method is called with argument *P*, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *key* be ? **ToPropertyKey**(*P*).
3. Repeat,
 - a. Let *desc* be ? *O*.**[[GetOwnProperty]]**(*key*).
 - b. If *desc* is not **undefined**, then
 - i. If **IsAccessorDescriptor**(*desc*) is **true**, return *desc*.**[[Get]]**.
 - ii. Return **undefined**.
 - c. Set *O* to ? *O*.**[[GetPrototypeOf]]**() .
 - d. If *O* is **null**, return **undefined**.

B.2.2.5 Object.prototype.__lookupSetter__ (*P*)

When the **__lookupSetter__** method is called with argument *P*, the following steps are taken:

1. Let *O* be ? **ToObject**(**this** value).
2. Let *key* be ? **ToPropertyKey**(*P*).
3. Repeat,
 - a. Let *desc* be ? *O*.**[[GetOwnProperty]]**(*key*).
 - b. If *desc* is not **undefined**, then
 - i. If **IsAccessorDescriptor**(*desc*) is **true**, return *desc*.**[[Set]]**.
 - ii. Return **undefined**.
 - c. Set *O* to ? *O*.**[[GetPrototypeOf]]**() .
 - d. If *O* is **null**, return **undefined**.

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** value to a String, starting from index *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 $size$ be the length of S .
4. Let $intStart$ be ? `ToIntegerOrInfinity`($start$).
5. If $intStart$ is $-\infty$, set $intStart$ to 0.
6. Else if $intStart < 0$, set $intStart$ to $\max(size + intStart, 0)$.
7. If $length$ is **undefined**, let $intLength$ be $size$; otherwise let $intLength$ be ? `ToIntegerOrInfinity`($length$).
8. If $intStart$ is $+\infty$, $intLength \leq 0$, or $intLength$ is $+\infty$, return the empty String.
9. Let $intEnd$ be $\min(intStart + intLength, size)$.
10. If $intStart \geq intEnd$, return the empty String.
11. Return the **substring** of S from $intStart$ to $intEnd$.

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*)

The abstract operation `CreateHTML` takes arguments *string*, *tag* (a String), *attribute* (a String), and *value*. It performs the following steps when called:

1. Let str be ? `RequireObjectCoercible`(*string*).
2. Let S be ? `ToString`(str).
3. Let $p1$ be the **string-concatenation** of "<" and *tag*.
4. If *attribute* is not the empty String, then
 - a. Let V be ? `ToString`(*value*).
 - b. Let $escapedV$ be the String value that is the same as V except that each occurrence of the code unit 0x0022 (QUOTATION MARK) in V has been replaced with the six code unit sequence """.
 - c. Set $p1$ to the **string-concatenation** of:
 - $p1$
 - the code unit 0x0020 (SPACE)
 - *attribute*
 - the code unit 0x003D (EQUALS SIGN)
 - the code unit 0x0022 (QUOTATION MARK)
 - $escapedV$
 - the code unit 0x0022 (QUOTATION MARK)
5. Let $p2$ be the **string-concatenation** of $p1$ and ">".
6. Let $p3$ be the **string-concatenation** of $p2$ and S .
7. Let $p4$ be the **string-concatenation** of $p3$, "</", *tag*, and ">".
8. Return $p4$.

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.2.3.15 String.prototype.trimLeft ()

NOTE The property **"trimStart"** is preferred. The **"trimLeft"** property is provided principally for compatibility with old code. It is recommended that the **"trimStart"** property be used in new ECMAScript code.

The initial value of the **"trimLeft"** property is the same [function object](#) as the initial value of the **String.prototype.trimStart** property.

B.2.3.16 String.prototype.trimRight ()

NOTE The property **"trimEnd"** is preferred. The **"trimRight"** property is provided principally for compatibility with old code. It is recommended that the **"trimEnd"** property be used in new ECMAScript code.

The initial value of the "**trimRight**" property is the same [function object](#) as the initial value of the **String.prototype.trimEnd** property.

B.2.4 Additional Properties of the Date.prototype Object

B.2.4.1 Date.prototype.getYear ()

NOTE The **getFullYear** method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the **getYear** method is called with no arguments, the following steps are taken:

1. Let *t* be ? [thisTimeValue](#)(**this** value).
2. If *t* is NaN, return NaN.
3. Return [YearFromTime](#)([LocalTime](#)(*t*)) - 1900_F.

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 ? [thisTimeValue](#)(**this** value).
2. If *t* is NaN, set *t* to +0_F; otherwise, set *t* to [LocalTime](#)(*t*).
3. Let *y* be ? [ToNumber](#)(*year*).
4. If *y* is NaN, then
 - a. Set the [[DateValue]] internal slot of [this Date object](#) to NaN.
 - b. Return NaN.
5. Let *yi* be ! [ToIntegerOrInfinity](#)(*y*).
6. If $0 \leq yi \leq 99$, let *yyyy* be 1900_F + $\mathbb{F}(yi)$.
7. Else, let *yyyy* be *y*.
8. Let *d* be [MakeDay](#)(*yyyy*, [MonthFromTime](#)(*t*), [DateFromTime](#)(*t*)).
9. Let *date* be [UTC](#)([MakeDate](#)(*d*, [TimeWithinDay](#)(*t*))).
10. Set the [[DateValue]] internal slot of [this Date object](#) to [TimeClip](#)(*date*).
11. Return the value of the [[DateValue]] internal slot of [this Date object](#).

B.2.4.3 Date.prototype.toGMTString ()

NOTE The **toUTCString** method is preferred. The **toGMTString** method is provided principally for compatibility with old code.

The [function object](#) that is the initial value of **Date.prototype.toGMTString** is the same [function object](#) that is the initial value of **Date.prototype.toUTCString**.

B.2.5 Additional Properties of the RegExp.prototype Object

B.2.5.1 RegExp.prototype.compile (*pattern*, *flags*)

When the **compile** method is called with arguments *pattern* and *flags*, the following steps are taken:

1. Let *O* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*O*, [[RegExpMatcher]]).
3. If [Type](#)(*pattern*) is Object and *pattern* has a [[RegExpMatcher]] internal slot, then
 - a. If *flags* is not **undefined**, throw a **TypeError** exception.
 - b. Let *P* be *pattern*.[[OriginalSource]].
 - c. Let *F* be *pattern*.[[OriginalFlags]].
4. Else,
 - a. Let *P* be *pattern*.
 - b. Let *F* be *flags*.
5. Return ? [RegExpInitialize](#)(*O*, *P*, *F*).

NOTE

The **compile** method completely reinitializes the **this** value 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 [13.2.6.1](#). This rule is **not** applied under any of the following circumstances:

- when *ObjectLiteral* appears in a context where *ObjectAssignmentPattern* is required,
- when initially parsing a *CoverParenthesizedExpressionAndArrowParameterList* or a *CoverCallExpressionAndAsyncArrowHead*, or
- when parsing text for `JSON.parse`.

ObjectLiteral :

```
{ PropertyDefinitionList }  
{ 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* .

NOTE

The [List](#) returned by [PropertyNameList](#) does not include string literal property names defined as using a *ComputedPropertyName*.

In [13.2.6.5](#) the [PropertyDefinitionEvaluation](#) algorithm for the production

PropertyDefinition : *PropertyName* : *AssignmentExpression*
is replaced with the following definition:

PropertyDefinition : *PropertyName* : *AssignmentExpression*

1. Let *propKey* be the result of evaluating *PropertyName*.
2. **ReturnIfAbrupt**(*propKey*).
3. If *propKey* is the String value "**__proto__**" and if **IsComputedPropertyKey**(*PropertyName*) is **false**, then
 - a. Let *isProtoSetter* be **true**.
4. Else,
 - a. Let *isProtoSetter* be **false**.
5. If **IsAnonymousFunctionDefinition**(*AssignmentExpression*) is **true** and *isProtoSetter* is **false**, then
 - a. Let *propValue* be ? **NamedEvaluation** of *AssignmentExpression* with argument *propKey*.
6. Else,
 - a. Let *exprValueRef* be the result of evaluating *AssignmentExpression*.
 - b. Let *propValue* be ? **GetValue**(*exprValueRef*).
7. If *isProtoSetter* is **true**, then
 - a. If **Type**(*propValue*) is either Object or Null, then
 - i. Return *object*.[[SetPrototypeOf]](*propValue*).
 - b. Return **NormalCompletion**(empty).
8. **Assert**: *enumerable* is **true**.
9. **Assert**: *object* is an ordinary, extensible object with no non-configurable properties.
10. Return ! **CreateDataPropertyOrThrow**(*object*, *propKey*, *propValue*).

B.3.2 Labelled Function Declarations

Prior to ECMAScript 2015, the specification of *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 code** and most browser-hosted ECMAScript implementations supported that extension. In ECMAScript 2015 and later, the grammar production for *LabelledStatement* permits use of *FunctionDeclaration* as a *LabelledItem* but 14.13.1 includes an Early Error rule that produces a Syntax Error if that occurs. That rule is modified with the addition of the highlighted text:

LabelledItem : *FunctionDeclaration*

- It is a Syntax Error if any strict mode source code matches this rule.

NOTE

The **early error** rules for *WithStatement*, *IfStatement*, and *IterationStatement* prevent these statements from containing a labelled *FunctionDeclaration* in **non-strict code**.

B.3.3 Block-Level Function Declarations Web Legacy Compatibility Semantics

Prior to ECMAScript 2015, 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

- One or more *FunctionDeclarations* whose *BindingIdentifier* is the name *f* occur 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 occurrences of *f* as an *IdentifierReference* are 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*.
- One or more *FunctionDeclarations* whose *BindingIdentifier* is the name *f* occur 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*
 - There may be occurrences of *f* as an *IdentifierReference* within the *StatementList* of the *Block* containing the declaration of *f*.
 - There is at least one occurrence of *f* as an *IdentifierReference* 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.
3. A function is declared and possibly used within a single block but also referenced within subsequent blocks.
- One or more *FunctionDeclaration* whose *BindingIdentifier* is the name *f* occur 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*
 - There may be occurrences of *f* as an *IdentifierReference* within the *StatementList* of the *Block* containing the declaration of *f*.
 - There is at least one occurrence of *f* as an *IdentifierReference* within the function code of *g* that lexically follows the *Block* containing the declaration of *f*.

The first use case is interoperable with the semantics of *Block* level function declarations provided by ECMAScript 2015. Any pre-existing ECMAScript code that employs that use case will operate using the *Block* level function declarations semantics defined by clauses 10, 14, and 15.

ECMAScript 2015 interoperability for the second and third use cases requires the following extensions to the clause 10, clause 15, clause 19.2.1 and clause 16.1.7 semantics.

If an ECMAScript implementation has a mechanism for reporting diagnostic warning messages, a warning should be produced when code contains a *FunctionDeclaration* for which these compatibility semantics are applied and introduce observable differences from non-compatibility semantics. For example, if a *var* binding is not introduced because its introduction would create an **early error**, a warning message should not be produced.

B.3.3.1 Changes to *FunctionDeclarationInstantiation*

During *FunctionDeclarationInstantiation* the following steps are performed in place of step 29:

29. If *strict* is **false**, then
- a. For each *FunctionDeclaration* *f* that is directly contained in the *StatementList* of a *Block*, *CaseClause*, or *DefaultClause*, do
 - i. Let *F* be **StringValue** of the *BindingIdentifier* of *f*.
 - ii. If replacing the *FunctionDeclaration* *f* with a *VariableStatement* that has *F* as a *BindingIdentifier* would not produce any Early Errors for *func* and *F* is not an element of *parameterNames*, then

1. NOTE: A var binding for *F* is only instantiated here if it is neither a *VarDeclaredName*, the name of a formal parameter, or another *FunctionDeclaration*.
2. If *initializedBindings* does not contain *F* and *F* is not "arguments", then
 - a. Perform ! *varEnv*.CreateMutableBinding(*F*, false).
 - b. Perform *varEnv*.InitializeBinding(*F*, undefined).
 - c. Append *F* to *instantiatedVarNames*.
3. When the *FunctionDeclaration* *f* is evaluated, perform the following steps in place of the *FunctionDeclaration* Evaluation algorithm provided in 15.2.6:
 - a. Let *fenv* be the running execution context's VariableEnvironment.
 - b. Let *benv* be the running execution context's LexicalEnvironment.
 - c. Let *fobj* be ! *benv*.GetBindingValue(*F*, false).
 - d. Perform ! *fenv*.SetMutableBinding(*F*, *fobj*, false).
 - e. Return NormalCompletion(empty).

B.3.3.2 Changes to GlobalDeclarationInstantiation

During *GlobalDeclarationInstantiation* the following steps are performed in place of step 13:

13. Let *strict* be IsStrict of *script*.
14. If *strict* is false, then
 - a. Let *declaredFunctionOrVarNames* be a new empty List.
 - b. Append to *declaredFunctionOrVarNames* the elements of *declaredFunctionNames*.
 - c. Append to *declaredFunctionOrVarNames* the elements of *declaredVarNames*.
 - d. For each *FunctionDeclaration* *f* that is directly contained in the *StatementList* of a *Block*, *CaseClause*, or *DefaultClause* Contained within *script*, do
 - i. Let *F* be StringValue of the *BindingIdentifier* of *f*.
 - ii. If replacing the *FunctionDeclaration* *f* with a *VariableStatement* that has *F* as a *BindingIdentifier* would not produce any Early Errors for *script*, then
 1. If *env*.HasLexicalDeclaration(*F*) is false, then
 - a. Let *fnDefinable* be ? *env*.CanDeclareGlobalVar(*F*).
 - b. If *fnDefinable* is true, then
 - i. NOTE: A var binding for *F* is only instantiated here if it is neither a *VarDeclaredName* nor the name of another *FunctionDeclaration*.
 - ii. If *declaredFunctionOrVarNames* does not contain *F*, then
 - i. Perform ? *env*.CreateGlobalVarBinding(*F*, false).
 - ii. Append *F* to *declaredFunctionOrVarNames*.
 - iii. When the *FunctionDeclaration* *f* is evaluated, perform the following steps in place of the *FunctionDeclaration* Evaluation algorithm provided in 15.2.6:
 - i. Let *genv* be the running execution context's VariableEnvironment.
 - ii. Let *benv* be the running execution context's LexicalEnvironment.
 - iii. Let *fobj* be ! *benv*.GetBindingValue(*F*, false).
 - iv. Perform ? *genv*.SetMutableBinding(*F*, *fobj*, false).
 - v. Return NormalCompletion(empty).

B.3.3.3 Changes to EvalDeclarationInstantiation

During *EvalDeclarationInstantiation* the following steps are performed in place of step 7:

7. If *strict* is false, then

- a. Let *declaredFunctionOrVarNames* be a new empty **List**.
- b. Append to *declaredFunctionOrVarNames* the elements of *declaredFunctionNames*.
- c. Append to *declaredFunctionOrVarNames* the elements of *declaredVarNames*.
- d. For each *FunctionDeclaration* *f* that is directly contained in the *StatementList* of a *Block*, *CaseClause*, or *DefaultClause* Contained within *body*, do
 - i. Let *F* be **StringValue** of the *BindingIdentifier* of *f*.
 - ii. If replacing the *FunctionDeclaration* *f* with a *VariableStatement* that has *F* as a *BindingIdentifier* would not produce any Early Errors for *body*, then
 1. Let *bindingExists* be **false**.
 2. Let *thisEnv* be *lexEnv*.
 3. **Assert**: The following loop will terminate.
 4. Repeat, while *thisEnv* is not the same as *varEnv*,
 - a. If *thisEnv* is not an **object Environment Record**, then
 - i. If *thisEnv*.HasBinding(*F*) is **true**, then
 - i. Let *bindingExists* be **true**.
 - b. Set *thisEnv* to *thisEnv*.[[OuterEnv]].
 5. If *bindingExists* is **false** and *varEnv* is a **global Environment Record**, then
 - a. If *varEnv*.HasLexicalDeclaration(*F*) is **false**, then
 - i. Let *fnDefinable* be ? *varEnv*.CanDeclareGlobalVar(*F*).
 - b. Else,
 - i. Let *fnDefinable* be **false**.
 6. Else,
 - a. Let *fnDefinable* be **true**.
 7. If *bindingExists* is **false** and *fnDefinable* is **true**, then
 - a. If *declaredFunctionOrVarNames* does not contain *F*, then
 - i. If *varEnv* is a **global Environment Record**, then
 - i. Perform ? *varEnv*.CreateGlobalVarBinding(*F*, **true**).
 - ii. Else,
 - i. Let *bindingExists* be *varEnv*.HasBinding(*F*).
 - ii. If *bindingExists* is **false**, then
 - i. Perform ! *varEnv*.CreateMutableBinding(*F*, **true**).
 - ii. Perform ! *varEnv*.InitializeBinding(*F*, **undefined**).
 - iii. Append *F* to *declaredFunctionOrVarNames*.
 - b. When the *FunctionDeclaration* *f* is evaluated, perform the following steps in place of the *FunctionDeclaration* Evaluation algorithm provided in 15.2.6:
 - i. Let *genv* be the **running execution context**'s VariableEnvironment.
 - ii. Let *benv* be the **running execution context**'s LexicalEnvironment.
 - iii. Let *fobj* be ! *benv*.GetBindingValue(*F*, **false**).
 - iv. Perform ? *genv*.SetMutableBinding(*F*, *fobj*, **false**).
 - v. Return **NormalCompletion**(empty).

B.3.3.4 Changes to Block Static Semantics: Early Errors

The rules for the following production in 14.2.1 are modified with the addition of the highlighted text:

Block : { *StatementList* }

- It is a Syntax Error if the **LexicallyDeclaredNames** of *StatementList* contains any duplicate entries, unless the

source code matching this production is not [strict mode code](#) and the duplicate entries are only bound by [FunctionDeclarations](#).

- It is a Syntax Error if any element of the [LexicallyDeclaredNames](#) of *StatementList* also occurs in the [VarDeclaredNames](#) of *StatementList*.

B.3.3.5 Changes to `switch` Statement Static Semantics: Early Errors

The rules for the following production in [14.12.1](#) are modified with the addition of the [highlighted](#) text:

SwitchStatement : **switch** (*Expression*) *CaseBlock*

- It is a Syntax Error if the [LexicallyDeclaredNames](#) of *CaseBlock* contains any duplicate entries, [unless the source code matching this production is not strict mode code and the duplicate entries are only bound by FunctionDeclarations](#).
- It is a Syntax Error if any element of the [LexicallyDeclaredNames](#) of *CaseBlock* also occurs in the [VarDeclaredNames](#) of *CaseBlock*.

B.3.3.6 Changes to `BlockDeclarationInstantiation`

During [BlockDeclarationInstantiation](#) the following steps are performed in place of step [3.a.ii.1](#):

1. If *env*.HasBinding(*dn*) is **false**, then
 - a. Perform !*env*.CreateMutableBinding(*dn*, **false**).

During [BlockDeclarationInstantiation](#) the following steps are performed in place of step [3.b.iii](#):

- iii. If the binding for *fn* in *env* is an uninitialized binding, then
 1. Perform *env*.InitializeBinding(*fn*, *fo*).
- iv. Else,
 1. **Assert**: *d* is a *FunctionDeclaration*.
 2. Perform *env*.SetMutableBinding(*fn*, *fo*, **false**).

B.3.4 FunctionDeclarations in IfStatement Statement Clauses

The following augments the *IfStatement* production in [14.6](#):

```
IfStatement[Yield, Await, Return] :  
  if ( Expression[+In, ?Yield, ?Await] ) FunctionDeclaration[?Yield, ?Await, ~Default] else  
    Statement[?Yield, ?Await, ?Return]  
  if ( Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return] else  
    FunctionDeclaration[?Yield, ?Await, ~Default]  
  if ( Expression[+In, ?Yield, ?Await] ) FunctionDeclaration[?Yield, ?Await, ~Default] else  
    FunctionDeclaration[?Yield, ?Await, ~Default]  
  if ( Expression[+In, ?Yield, ?Await] ) FunctionDeclaration[?Yield, ?Await, ~Default] [lookahead ≠ else]
```

This production only applies when parsing [non-strict code](#). Code matching this production is processed as if each matching occurrence of *FunctionDeclaration*_[?Yield, ?Await, ~Default] 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 14.15.1 is replaced with the following:

Catch : **catch** (*CatchParameter*) *Block*

- It is a Syntax Error if **BoundNames** of *CatchParameter* contains any duplicate elements.
- 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 *CatchParameter* is *CatchParameter* : *BindingIdentifier* .

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* for such **var** declarations will assign to the corresponding catch parameter rather than the **var** binding.

This modified behaviour also applies to **var** and **function** declarations introduced by **direct eval** calls contained within the *Block* of a *Catch* clause. This change is accomplished by modifying the algorithm of 19.2.1.3 as follows:

Step 3.d.i.2.a.i is replaced by:

- i. If *thisEnv* is not the **Environment Record** for a *Catch* clause, throw a **SyntaxError** exception.

Step 7.d.ii.4.a.i.i is replaced by:

- i. If *thisEnv* is not the **Environment Record** for a *Catch* clause, let *bindingExists* be **true**.

B.3.6 Initializers in ForIn Statement Heads

The following augments the *ForInOfStatement* production in 14.7.5:

```
ForInOfStatement[Yield, Await, Return] :  
  for ( var BindingIdentifier[?Yield, ?Await] Initializer[~In, ?Yield, ?Await] in  
    Expression[+In, ?Yield, ?Await] ) Statement[?Yield, ?Await, ?Return]
```

This production only applies when parsing **non-strict code**.

The **static semantics** of **ContainsDuplicateLabels** in 8.2.1 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Return **ContainsDuplicateLabels** of *Statement* with argument *labelSet*.

The **static semantics** of **ContainsUndefinedBreakTarget** in 8.2.2 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Return **ContainsUndefinedBreakTarget** of *Statement* with argument *labelSet*.

The **static semantics** of **ContainsUndefinedContinueTarget** in 8.2.3 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Return [ContainsUndefinedContinueTarget](#) of *Statement* with arguments *iterationSet* and « ».

The [static semantics](#) of [IsDestructuring](#) in 14.7.5.2 are augmented with the following:

BindingIdentifier :

Identifier

yield

await

1. Return **false**.

The [static semantics](#) of [VarDeclaredNames](#) in 8.1.6 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Let *names* be the [BoundNames](#) of *BindingIdentifier*.
2. Append to *names* the elements of the [VarDeclaredNames](#) of *Statement*.
3. Return *names*.

The [static semantics](#) of [VarScopedDeclarations](#) in 8.1.7 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Let *declarations* be a [List](#) whose sole element is *BindingIdentifier*.
2. Append to *declarations* the elements of the [VarScopedDeclarations](#) of *Statement*.
3. Return *declarations*.

The [runtime semantics](#) of [ForInOfLoopEvaluation](#) in 14.7.5.5 are augmented with the following:

ForInOfStatement : **for** (**var** *BindingIdentifier* *Initializer* **in** *Expression*) *Statement*

1. Let *bindingId* be [StringValue](#) of *BindingIdentifier*.
2. Let *lhs* be ? [ResolveBinding](#)(*bindingId*).
3. If [IsAnonymousFunctionDefinition](#)(*Initializer*) is **true**, then
 - a. Let *value* be [NamedEvaluation](#) of *Initializer* with argument *bindingId*.
4. Else,
 - a. Let *rhs* be the result of evaluating *Initializer*.
 - b. Let *value* be ? [GetValue](#)(*rhs*).
5. Perform ? [PutValue](#)(*lhs*, *value*).
6. Let *keyResult* be ? [ForIn/OfHeadEvaluation](#)(« », *Expression*, **enumerate**).
7. Return ? [ForIn/OfBodyEvaluation](#)(*BindingIdentifier*, *Statement*, *keyResult*, **enumerate**, *varBinding*, *labelSet*).

B.3.7 The [\[\[IsHTMLDDA\]\]](#) Internal Slot

An [\[\[IsHTMLDDA\]\]](#) *internal slot* may exist on [host-defined](#) objects. Objects with an [\[\[IsHTMLDDA\]\]](#) *internal slot* behave like **undefined** in the [ToBoolean](#) and [Abstract Equality Comparison](#) abstract operations and when used as an operand for the **typeof** operator.

NOTE

Objects with an `[[IsHTMLDDA]]` internal slot are never created by this specification. However, the `document.all` object in web browsers is a *host-defined exotic object* with this slot that exists for web compatibility purposes. There are no other known examples of this type of object and implementations should not create any with the exception of `document.all`.

B.3.7.1 Changes to ToBoolean

The result column in [Table 11](#) for an argument type of Object is replaced with the following algorithm:

1. If *argument* has an `[[IsHTMLDDA]]` internal slot, return **false**.
2. Return **true**.

B.3.7.2 Changes to Abstract Equality Comparison

The following steps replace step 4 of the *Abstract Equality Comparison* algorithm:

1. If `Type(x)` is Object and *x* has an `[[IsHTMLDDA]]` internal slot and *y* is either **null** or **undefined**, return **true**.
2. If *x* is either **null** or **undefined** and `Type(y)` is Object and *y* has an `[[IsHTMLDDA]]` internal slot, return **true**.

B.3.7.3 Changes to the typeof Operator

The following table entry is inserted into [Table 37](#) immediately preceding the entry for "Object (implements `[[Call]]`)":

Table 83: Additional **typeof** Operator Results

Type of <i>val</i>	Result
Object (has an <code>[[IsHTMLDDA]]</code> internal slot)	"undefined"

C The Strict Mode of ECMAScript

The strict mode restriction and exceptions

- **implements**, **interface**, **let**, **package**, **private**, **protected**, **public**, **static**, and **yield** are reserved words within *strict mode code*. (12.6.2).
- A conforming implementation, when processing *strict mode code*, must not extend, as described in B.1.1, the syntax of *NumericLiteral* to include *LegacyOctalIntegerLiteral*, nor extend the syntax of *DecimalIntegerLiteral* to include *NonOctalDecimalIntegerLiteral*.
- A conforming implementation, when processing *strict mode code*, may not extend the syntax of *EscapeSequence* to include *LegacyOctalEscapeSequence* or *NonOctalDecimalEscapeSequence* 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 *LeftHandSideExpression* must not evaluate to an unresolvable Reference. If it does a **ReferenceError** exception is thrown (6.2.4.5). The *LeftHandSideExpression* also may not be a reference to a **data property** with the attribute value { `[[Writable]]`: **false** }, to an **accessor property** with the attribute value { `[[Set]]`: **undefined** }, nor to a non-existent property of an object whose `[[Extensible]]` internal slot has the value **false**. In these cases a **TypeError** exception is thrown (13.15).

- An *IdentifierReference* with the [StringValue](#) "eval" or "arguments" may not appear as the *LeftHandSideExpression* of an Assignment operator (13.15) or of an *UpdateExpression* (13.4) or as the *UnaryExpression* operated upon by a Prefix Increment (13.4.4) or a Prefix Decrement (13.4.5) operator.
- Arguments objects for strict functions define a non-configurable [accessor property](#) "callee" which throws a **TypeError** exception on access (10.4.4.6).
- Arguments objects for strict functions do not dynamically share their [array-indexed](#) property values with the corresponding formal parameter bindings of their functions. (10.4.4).
- For strict functions, if an arguments object is created the binding of the local identifier **arguments** to the arguments object is immutable and hence may not be the target of an assignment expression. (10.2.10).
- It is a **SyntaxError** if the [StringValue](#) of a *BindingIdentifier* is "eval" or "arguments" within [strict mode code](#) (13.1.1).
- 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 (19.2.1).
- If **this** is evaluated within [strict mode code](#), then the **this** value is not coerced to an object. A **this** value of **undefined** or **null** is not converted to the [global object](#) and primitive values are not converted to wrapper objects. The **this** value passed via a function call (including calls made using **Function.prototype.apply** and **Function.prototype.call**) do not coerce the passed **this** value to an object (10.2.1.2, 20.2.3.1, 20.2.3.3).
- 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 (13.5.1.1).
- When a **delete** operator occurs within [strict mode code](#), a **TypeError** is thrown if the property to be deleted has the attribute { [[Configurable]]: **false** } or otherwise cannot be deleted (13.5.1.2).
- [Strict mode code](#) may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context is a **SyntaxError** (14.11.1).
- It is a **SyntaxError** if a *CatchParameter* occurs within [strict mode code](#) and [BoundNames](#) of *CatchParameter* contains either **eval** or **arguments** (14.15.1).
- It is a **SyntaxError** if the same *BindingIdentifier* appears more than once in the *FormalParameters* of a [strict function](#). An attempt to create such a function using a Function, Generator, or AsyncFunction [constructor](#) is a **SyntaxError** (15.2.1, 20.2.1.1.1).
- An implementation may not extend, beyond that defined in this specification, the meanings within strict functions of properties named "caller" or "arguments" of function instances.

D Host Layering Points

See 4.2 for the definition of [host](#).

D.1 Host Hooks

[HostCallJobCallback\(...\)](#)

[HostEnqueueFinalizationRegistryCleanupJob\(...\)](#)

[HostEnqueuePromiseJob\(...\)](#)

[HostEnsureCanCompileStrings\(...\)](#)

[HostFinalizeImportMeta\(...\)](#)

[HostGetImportMetaProperties\(...\)](#)

[HostHasSourceTextAvailable\(...\)](#)

[HostImportModuleDynamically\(...\)](#)

[HostMakeJobCallback\(...\)](#)

[HostPromiseRejectionTracker\(...\)](#)

[HostResolveImportedModule\(...\)](#)

[InitializeHostDefinedRealm\(...\)](#)

D.2 Host-defined Fields

[[HostDefined]] on [Realm](#) Records: See [Table 23](#).

[[HostDefined]] on [Script](#) Records: See [Table 39](#).

[[HostDefined]] on [Module](#) Records: See [Table 40](#).

[[HostDefined]] on [JobCallback](#) Records: See [Table 27](#).

[[HostSynchronizesWith]] on [Candidate Executions](#): See [Table 81](#).

[[IsHTMLDDA]]: See [B.3.7](#).

D.3 Host-defined Objects

The [global object](#): See [clause 19](#).

D.4 Running Jobs

Preparation steps before, and cleanup steps after, invocation of [Job](#) Abstract Closures. See [9.4](#).

D.5 Internal Methods of Exotic Objects

Any of the essential internal methods in [Table 6](#) for any [exotic object](#) not specified within this specification.

D.6 Built-in Objects and Methods

Any built-in objects and methods not defined within this specification, except as restricted in [17.1](#).

E Corrections and Clarifications in ECMAScript 2015 with Possible Compatibility Impact

[9.1.1.4.15-9.1.1.4.18](#) Edition 5 and 5.1 used a property existence test to determine whether a [global object](#) property corresponding to a new global declaration already existed. ECMAScript 2015 uses an own property existence test. This corresponds to what has been most commonly implemented by web browsers.

[10.4.2.1](#): 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. ECMAScript 2015 specifies that the current array length must be captured after the possible occurrence of such side-effects.

[21.4.1.14](#): Previous editions permitted the [TimeClip](#) abstract operation to return either $+0_{\text{F}}$ or -0_{F} as the representation of a 0 [time value](#). ECMAScript 2015 specifies that $+0_{\text{F}}$ always returned. This means that for ECMAScript 2015 the [time value](#) of a Date object is never observably -0_{F} and methods that return time values never return -0_{F} .

[21.4.1.15](#): If a UTC offset representation is not present, the local time zone is used. Edition 5.1 incorrectly stated that a missing time zone should be interpreted as "z".

[21.4.4.36](#): If the year cannot be represented using the Date Time String Format specified in [21.4.1.15](#) a RangeError exception is thrown. Previous editions did not specify the behaviour for that case.

[21.4.4.41](#): Previous editions did not specify the value returned by [Date.prototype.toString](#) when [this time value](#) is NaN. ECMAScript 2015 specifies the result to be the String value "Invalid Date".

[22.2.3.1](#), [22.2.3.2.5](#): Any LineTerminator code points in the value of the "source" property of a RegExp instance must be expressed using an escape sequence. Edition 5.1 only required the escaping of `/`.

[22.2.5.7](#), [22.2.5.10](#): In previous editions, the specifications for [String.prototype.match](#) and [String.prototype.replace](#) was incorrect for cases where the pattern argument was a RegExp value whose [global](#) flag is set. The previous specifications stated that for each attempt to match the pattern, if [lastIndex](#) did not change it should be incremented by 1. The correct behaviour is that [lastIndex](#) should be incremented by one only if the pattern matched the empty String.

[23.1.3.27](#), [23.1.3.27.1](#): Previous editions did not specify how a NaN value returned by a [comparefn](#) was interpreted by [Array.prototype.sort](#). ECMAScript 2015 specifies that such a value is treated as if $+0_{\text{F}}$ was returned from the [comparefn](#). ECMAScript 2015 also specifies that [ToNumber](#) is applied to the result returned by a [comparefn](#). In previous editions, the effect of a [comparefn](#) result that is not a [Number value](#) was [implementation-defined](#). In practice, implementations call [ToNumber](#).

F Additions and Changes That Introduce Incompatibilities with Prior Editions

[6.2.4](#): In ECMAScript 2015, Function calls are not allowed to return a [Reference Record](#).

[7.1.4.1](#): In ECMAScript 2015, [ToNumber](#) applied to a String value now recognizes and converts *BinaryIntegerLiteral* and *OctalIntegerLiteral* numeric strings. In previous editions such strings were converted to NaN.

9.2: In ECMAScript 2018, Template objects are canonicalized based on [Parse Node](#) (source location), instead of across all occurrences of that template literal or tagged template in a [Realm](#) in previous editions.

12.2: In ECMAScript 2016, Unicode 8.0.0 or higher is mandated, as opposed to ECMAScript 2015 which mandated Unicode 5.1. In particular, this caused U+180E MONGOLIAN VOWEL SEPARATOR, which was in the **Space_Separator (Zs)** category and thus treated as whitespace in ECMAScript 2015, to be moved to the **Format (Cf)** category (as of Unicode 6.3.0). This causes whitespace-sensitive methods to behave differently. For example, `"\u180E".trim().length` was **0** in previous editions, but **1** in ECMAScript 2016 and later. Additionally, ECMAScript 2017 mandated always using the latest version of the Unicode standard.

12.6: In ECMAScript 2015, the valid code points for an *IdentifierName* are specified in terms of the Unicode properties “ID_Start” and “ID_Continue”. In previous editions, the valid *IdentifierName* or *Identifier* code points were specified by enumerating various Unicode code point categories.

12.9.1: In ECMAScript 2015, Automatic Semicolon Insertion adds a semicolon at the end of a do-while statement if the semicolon is missing. This change aligns the specification with the actual behaviour of most existing implementations.

13.2.6.1: In ECMAScript 2015, it is no longer an [early error](#) to have duplicate property names in Object Initializers.

13.15.1: In ECMAScript 2015, [strict mode code](#) containing an assignment to an immutable binding such as the function name of a *FunctionExpression* does not produce an [early error](#). Instead it produces a runtime error.

14.2: In ECMAScript 2015, a *StatementList* beginning with the token `let` followed by the input elements *LineTerminator* then *Identifier* is the start of a *LexicalDeclaration*. In previous editions, automatic semicolon insertion would always insert a semicolon before the *Identifier* input element.

14.5: In ECMAScript 2015, a *StatementListItem* beginning with the token **let** followed by the token `[` is the start of a *LexicalDeclaration*. In previous editions such a sequence would be the start of an *ExpressionStatement*.

14.6.2: In ECMAScript 2015, the normal completion value of an *IfStatement* is never the value **empty**. If no *Statement* part is evaluated or if the evaluated *Statement* part produces a normal completion whose value is **empty**, the completion value of the *IfStatement* is **undefined**.

14.7: In ECMAScript 2015, if the **(** token of a for statement is immediately followed by the token sequence **let [** then the **let** is treated as the start of a *LexicalDeclaration*. In previous editions such a token sequence would be the start of an *Expression*.

14.7: In ECMAScript 2015, if the **(** token of a for-in statement is immediately followed by the token sequence **let [** then the **let** is treated as the start of a *ForDeclaration*. In previous editions such a token sequence would be the start of an *LeftHandSideExpression*.

14.7: Prior to ECMAScript 2015, an initialization expression could appear as part of the *VariableDeclaration* that precedes the **in** keyword. In ECMAScript 2015, the *ForBinding* in that same position does not allow the occurrence of such an initializer. In ECMAScript 2017, such an initializer is permitted only in [non-strict code](#).

14.7: In ECMAScript 2015, the completion value of an *IterationStatement* is never the value **empty**. If the *Statement* part of an *IterationStatement* is not evaluated or if the final evaluation of the *Statement* part produces a completion whose value is **empty**, the completion value of the *IterationStatement* is **undefined**.

14.11.2: In ECMAScript 2015, the normal completion value of a *WithStatement* is never the value **empty**. If evaluation of the *Statement* part of a *WithStatement* produces a normal completion whose value is **empty**, the completion value of the

WithStatement is **undefined**.

14.12.4: In ECMAScript 2015, the completion value of a *SwitchStatement* is never the value **empty**. If the *CaseBlock* part of a *SwitchStatement* produces a completion whose value is **empty**, the completion value of the *SwitchStatement* is **undefined**.

14.15: In ECMAScript 2015, 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.

14.15, 19.2.1.3: In ECMAScript 2015, a runtime **SyntaxError** is thrown if a *Catch* clause evaluates a non-strict direct **eval** whose eval code includes a **var** or **FunctionDeclaration** declaration that binds the same *Identifier* that appears as the *Catch* clause parameter.

14.15.3: In ECMAScript 2015, the completion value of a *TryStatement* is never the value **empty**. If the *Block* part of a *TryStatement* evaluates to a normal completion whose value is **empty**, the completion value of the *TryStatement* is **undefined**. If the *Block* part of a *TryStatement* evaluates to a throw completion and it has a *Catch* part that evaluates to a normal completion whose value is **empty**, the completion value of the *TryStatement* is **undefined** if there is no *Finally* clause or if its *Finally* clause evaluates to an **empty** normal completion.

15.4.5 In ECMAScript 2015, 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 and they do not have a **"prototype"** own property. In the previous edition, they were constructors and had a **"prototype"** property.

20.1.2.6: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.8: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.10: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.12: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.14: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.15: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.16: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.17: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.18: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.1.2.20: In ECMAScript 2015, 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 the previous edition, a non-object argument always causes a **TypeError** to be thrown.

20.2.3.2: In ECMAScript 2015, the `[[Prototype]]` internal slot of a bound function is set to the `[[GetPrototypeOf]]` value of its target function. In the previous edition, `[[Prototype]]` was always set to `%Function.prototype%`.

20.2.4.1: In ECMAScript 2015, the **"length"** property of function instances is configurable. In previous editions it was non-configurable.

20.5.6.2: In ECMAScript 2015, the `[[Prototype]]` internal slot of a **NativeError constructor** is the **Error constructor**. In previous editions it was the **Function prototype object**.

21.4.4 In ECMAScript 2015, the **Date prototype object** is not a Date instance. In previous editions it was a Date instance whose `TimeValue` was `NaN`.

22.1.3.10 In ECMAScript 2015, the **String.prototype.localeCompare** function must treat Strings that are canonically equivalent according to the Unicode standard as being identical. In previous editions implementations were permitted to ignore canonical equivalence and could instead use a bit-wise comparison.

22.1.3.26 and 22.1.3.28 In ECMAScript 2015, 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.

22.1.3.29 In ECMAScript 2015, 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 7 no such code points are defined. In previous editions such code points would not have been recognized as white space.

22.2.3.1 In ECMAScript 2015, If the *pattern* argument is a RegExp instance and the *flags* argument is not **undefined**, a new RegExp instance is created just like *pattern* except that *pattern*'s flags are replaced by the argument *flags*. In previous editions a **TypeError** exception was thrown when *pattern* was a RegExp instance and *flags* was not **undefined**.

22.2.5 In ECMAScript 2015, the **RegExp prototype object** is not a RegExp instance. In previous editions it was a RegExp instance whose pattern is the empty String.

22.2.5 In ECMAScript 2015, **"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.

25.4.12: In ECMAScript 2019, **Atomics.wake** has been renamed to **Atomics.notify** to prevent confusion with **Atomics.wait**.

27.1.4.4, 27.6.3.6: In ECMAScript 2019, the number of Jobs enqueued by **await** was reduced, which could create an observable difference in resolution order between a **then()** call and an **await** expression.

G Colophon

This specification is authored on [GitHub](#) in a plaintext source format called [Ecm Markup](#). Ecm Markup is an HTML and Markdown dialect that provides a framework and toolset for authoring ECMAScript specifications in plaintext and processing the specification into a full-featured HTML rendering that follows the editorial conventions for this document. Ecm Markup builds on and integrates a number of other formats and technologies including [Grammarkdown](#) for defining syntax and [Ecm Markupdown](#) for authoring algorithm steps. PDF renderings of this specification are produced by printing the HTML rendering to a PDF.

Prior editions of this specification were authored using Word—the Ecm Markup source text that formed the basis of this edition was produced by converting the ECMAScript 2015 Word document to Ecm Markup using an automated conversion tool.

H Bibliography

1. *IEEE 754-2019: IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronic Engineers, New York (2019)

NOTE

There are no normative changes between IEEE 754-2008 and IEEE 754-2019 that affect the ECMA-262 specification.

2. *The Unicode Standard*, available at <<https://unicode.org/versions/latest>>
3. *Unicode Technical Note #5: Canonical Equivalence in Applications*, available at <<https://unicode.org/notes/tn5/>>
4. *Unicode Technical Standard #10: Unicode Collation Algorithm*, available at <<https://unicode.org/reports/tr10/>>
5. *Unicode Standard Annex #15, Unicode Normalization Forms*, available at <<https://unicode.org/reports/tr15/>>
6. *Unicode Standard Annex #18: Unicode Regular Expressions*, available at <<https://unicode.org/reports/tr18/>>
7. *Unicode Standard Annex #24: Unicode **Script** Property*, available at <<https://unicode.org/reports/tr24/>>
8. *Unicode Standard Annex #31, Unicode Identifiers and Pattern Syntax*, available at <<https://unicode.org/reports/tr31/>>
9. *Unicode Standard Annex #44: Unicode Character Database*, available at <<https://unicode.org/reports/tr44/>>
10. *Unicode Technical Standard #51: Unicode Emoji*, available at <<https://unicode.org/reports/tr51/>>
11. *IANA Time Zone Database*, available at <<https://www.iana.org/time-zones>>
12. ISO 8601:2004(E) *Data elements and interchange formats — Information interchange — Representation of dates and times*
13. RFC 1738 “Uniform Resource Locators (URL)”, available at <<https://tools.ietf.org/html/rfc1738>>
14. RFC 2396 “Uniform Resource Identifiers (URI): Generic Syntax”, available at <<https://tools.ietf.org/html/rfc2396>>
15. RFC 3629 “UTF-8, a transformation format of ISO 10646”, available at <<https://tools.ietf.org/html/rfc3629>>
16. RFC 7231 “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content”, available at <<https://tools.ietf.org/html/rfc7231>>

I Copyright & Software License

Rue du Rhone 114

CH-1204 Geneva

Tel: +41 22 849 6000

Fax: +41 22 849 6001

Web: <https://ecma-international.org/>

Copyright Notice

© 2021 Ecma International

This document may be copied, published and distributed to others, and certain derivative works of it may be prepared, copied, published, and distributed, in whole or in part, provided that the above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:

- (i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),
- (ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,
- (iii) translations of this document into languages other than English and into different formats and
- (iv) works by making use of this specification in standard conformant products by implementing (e.g. by copy and paste wholly or partly) the functionality therein.

However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.

The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.

The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns.

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.

Software License

All Software contained in this document ("Software") is protected by copyright and is being made available under the "BSD License", included below. This Software may be subject to third party rights (rights from parties other than Ecma International), including patent rights, and no licenses under such third party rights are granted under this license

even if the third party concerned is a member of Ecma International. SEE THE ECMA CODE OF CONDUCT IN PATENT MATTERS AVAILABLE AT <https://ecma-international.org/memento/codeofconduct.htm> FOR INFORMATION REGARDING THE LICENSING OF PATENT CLAIMS THAT ARE REQUIRED TO IMPLEMENT ECMA INTERNATIONAL STANDARDS.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the authors nor Ecma International may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE ECMA INTERNATIONAL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ECMA INTERNATIONAL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.