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.
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 ( \textit{string}, \textit{searchValue}, \textit{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 ( \textit{x} )
6.1.6.1.2 Number::bitwiseNOT ( \textit{x} )
6.1.6.1.3 Number::exponentiate ( \textit{base}, \textit{exponent} )
6.1.6.1.4 Number::multiply ( \textit{x}, \textit{y} )
6.1.6.1.5 Number::divide ( \textit{x}, \textit{y} )
6.1.6.1.6 Number::remainder ( \textit{n}, \textit{d} )
6.1.6.1.7 Number::add ( \textit{x}, \textit{y} )
6.1.6.1.8 Number::subtract ( \textit{x}, \textit{y} )
6.1.6.1 Number::leftShift (x, y)
6.1.6.10 Number::signedRightShift (x, y)
6.1.6.11 Number::unsignedRightShift (x, y)
6.1.6.12 Number::lessThan (x, y)
6.1.6.13 Number::equal (x, y)
6.1.6.14 Number::sameValue (x, y)
6.1.6.15 Number::sameValueZero (x, y)
6.1.6.16 NumberBitwiseOp (op, x, y)
6.1.6.17 Number::bitwiseAND (x, y)
6.1.6.18 Number::bitwiseXOR (x, y)
6.1.6.19 Number::bitwiseOR (x, y)
6.1.6.20 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 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]] ()
10.1.12 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 \[\text{GetPrototypeOf}\] ( )
10.5.2 \[\text{SetPrototypeOf}\] ( \( V \) )
10.5.3 \[\text{IsExtensible}\] ( )
10.5.4 \[\text{PreventExtensions}\] ( )
10.5.5 \[\text{GetOwnProperty}\] ( \( P \) )
10.5.6 \[\text{DefineOwnProperty}\] ( \( P, \text{Desc} \) )
10.5.7 \[\text{HasProperty}\] ( \( P \) )
10.5.8 \[\text{Get}\] ( \( P, \text{Receiver} \) )
10.5.9 \[\text{Set}\] ( \( P, V, \text{Receiver} \) )
10.5.10 \[\text{Delete}\] ( \( P \) )
10.5.11 \[\text{OwnPropertyKeys}\] ( )
10.5.12 \[\text{Call}\] ( \( \text{thisArgument}, \text{argumentsList} \) )
10.5.13 \[\text{Construct}\] ( \( \text{argumentsList}, \text{newTarget} \) )
10.5.14 \text{ProxyCreate} ( \( \text{target}, \text{handler} \) )

11 ECMAScript Language: Source Code

11.1 Source Text

11.1.1 \text{UTF16EncodeCodePoint} ( \( cp \) )
11.1.2 \text{CodePointsToString} ( \text{text} )
11.1.3 \text{UTF16SurrogatePairToCodePoint} ( \text{lead, trail} )
11.1.4 \text{CodePointAt} ( \text{string, position} )
11.1.5 \text{StringToCodePoints} ( \text{string} )
11.1.6 \text{ParseText} ( \text{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 \text{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 \text{MV}
12.8.3.2 \text{NumericValue}
12.8.4 String Literals

12.8.4.1 \text{SV}
12.8.4.2 \text{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
13.14.1 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 (initializedBoundNames, 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
14.15 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 )
   16.2.2 Imports
      16.2.2.1 SS: Early Errors
      16.2.2.2 SS: ImportEntries
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 ObjectDefineProperties ( 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.getOwnPropertyNames ( O )
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 ( )
20.4.3.4 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.getUTCMilliseconds ( )
21.4.4.18 Date.prototype.getUTCMonth ( )
21.4.4.19 Date.prototype.getUTCSec ( )
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.setUTCMilliseconds ( ms )
21.4.4.33 Date.prototype.setUTCMonth ( month [ , date ] )
21.4.4.34 Date.prototype.setUTCSec ( sec [ , ms ] )
21.4.4.35 Date.prototype.toDateString ( )
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 String (value)

22.1.2 Properties of the String Constructor
22.1.2.1 String.fromCharCode (...codeUnits)
22.1.2.2 String.fromCodePoint (...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 RegEx (Regular Expression) Objects

22.2.1 Patterns

22.2.1.1 SS: Early Errors
22.2.1.2 SS: CapturingGroupNumber
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% Object

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 [ , mapper ] [ , 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.findIndex (predicate [, thisArg ])

23.2.3.11 `%TypedArray%.prototype.forEach (callbackfn [, thisArg ])

23.2.3.12 `%TypedArray%.prototype.includes (searchElement [, fromIndex ])

23.2.3.13 `%TypedArray%.prototype.indexOf (searchElement [, fromIndex ])

23.2.3.14 `%TypedArray%.prototype.join (separator)

23.2.3.15 `%TypedArray%.prototype.keys ()

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.findIndex (predicate [, thisArg ])

23.2.3.22 `%TypedArray%.prototype.reverse ()

23.2.3.23 `%TypedArray%.prototype.set (source [, offset ])

23.2.3.24 AllocateTypedArrayFromTypedArray (target, targetOffset, source)

23.2.3.25 AllocateTypedArrayFromArrayBuffer (O, arrayLike)

23.2.3.26 AllocateTypedArrayFromArrayLike (O, arrayLike)

23.2.3.27 AllocateTypedArrayFromArrayBuffer (O, buffer, byteOffset, length)

23.2.3.28 AllocateTypedArrayFromList (O, values)

23.2.3.29 AllocateTypedArrayFromTypedArray (O, arrayLike)

23.2.3.30 AllocateTypedArrayBuffer (O, length)

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`
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 ]

24.2.5.3 Set Iterator Objects

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 ( \textit{arrayBuffer}, \textit{byteIndex}, \textit{type}, \textit{value}, \textit{op} [, \textit{isLittleEndian}] )

25.1.3 The ArrayBuffer Constructor

25.1.3.1 ArrayBuffer ( \textit{length} )

25.1.4 Properties of the ArrayBuffer Constructor

25.1.4.1 ArrayBuffer.isView ( \textit{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 ( \textit{start}, \textit{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 ( \textit{constructor}, \textit{byteLength} )

25.2.1.2 IsSharedArrayBuffer ( \textit{obj} )

25.2.2 The SharedArrayBuffer Constructor

25.2.2.1 SharedArrayBuffer ( [ \textit{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 ( \textit{start}, \textit{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 ( \textit{view}, \textit{requestIndex}, \textit{isLittleEndian}, \textit{type} )

25.3.1.2 SetViewValue ( \textit{view}, \textit{requestIndex}, \textit{isLittleEndian}, \textit{type}, \textit{value} )

25.3.2 The DataView Constructor

25.3.2.1 DataView ( \textit{buffer} [, \textit{byteOffset} [, \textit{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 ( \textit{byteOffset} [, \textit{littleEndian} ] )

25.3.4.6 DataView.prototype.getBigUint64 ( \textit{byteOffset} [, \textit{littleEndian} ] )

25.3.4.7 DataView.prototype.getFloat32 ( \textit{byteOffset} [, \textit{littleEndian} ] )

25.3.4.8 DataView.prototype.getFloat64 ( \textit{byteOffset} [, \textit{littleEndian} ] )

25.3.4.9 DataView.prototype.getInt8 ( \textit{byteOffset} )

25.3.4.10 DataView.prototype.getInt16 ( \textit{byteOffset} [, \textit{littleEndian} ] )

25.3.4.11 DataView.prototype.getInt32 ( \textit{byteOffset} [, \textit{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)

25.3.5 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)

25.5 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 ] )
    27.1.4.4 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

27.2.4.3 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)

27.7 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)
Introduction

This Ecma Standard defines the ECMA-Script 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.


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


After publication of the third edition, ECMAScript achieved massive adoption in conjunction with the World Wide Web where it has become the programming language that is supported by essentially all web browsers. Significant work was done to develop a fourth edition of ECMAScript. 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.


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 Ecmarkup, Ecmarkdown, and Grammarkdown. 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 (\texttt{1_000}); and \texttt{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, 6\textsuperscript{th} Edition

Brian Terlson  
ECMA-262, Project Editor, 7\textsuperscript{th} through 10\textsuperscript{th} Editions

Jordan Harband  
ECMA-262, Project Editor, 10\textsuperscript{th} through 12\textsuperscript{th} 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.


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

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

**CF** is a **constructor** (and also an object). Five objects have been created by using **new** expressions: $cf_1$, $cf_2$, $cf_3$, $cf_4$, and $cf_5$. Each of these objects contains properties named "$q1$" and "$q2$". The dashed lines represent the implicit prototype relationship; so, for example, $cf_3$’s prototype is $CF_p$. The **constructor**, **CF**, has two properties itself, named "$P1$" and "$P2$", which are not visible to $CF_p$, $cf_1$, $cf_2$, $cf_3$, $cf_4$, or $cf_5$. The property named "$CFP1$" in $CF_p$ is shared by $cf_1$, $cf_2$, $cf_3$, $cf_4$, and $cf_5$ (but not by **CF**), as are any properties found in $CF_p$’s implicit prototype chain that are not named "$q1$", "$q2$", or "$CFP1$". Notice that there is no implicit prototype link between **CF** and $CF_p$.

Unlike 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 $cf_1$, $cf_2$, $cf_3$, $cf_4$, and $cf_5$ 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**
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:

```javascript
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:

\[\text{WhileStatement :}
\]
\[\text{\quad while ( Expression ) Statement}\]

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:

\[\text{ArgumentList :}
\]
\[\text{\quad AssignmentExpression}
\]
\[\text{\quad ArgumentList , AssignmentExpression}\]

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:

\[\text{VariableDeclaration :}
\]
\[\text{\quad BindingIdentifier \ Initializer_{opt}}\]

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:

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

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

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

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

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

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

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

is equivalent to saying:

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

and:

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

is equivalent to:

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

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

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

is an abbreviation for:
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:

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

which is merely a convenient abbreviation for:

\[
\text{NonZeroDigit} ::  \\
\quad 1  \\
\quad 2  \\
\quad 3  \\
\quad 4  \\
\quad 5  \\
\quad 6  \\
\quad 7  \\
\quad 8  \\
\quad 9
\]

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:

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

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

the definition:

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

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

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:

\[
\text{ThrowStatement} : \quad \text{throw} \; [\text{no LineTerminator here}] \; \text{Expression} ;
\]

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:

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

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:

\[
\text{SourceCharacter} :: \quad \text{any Unicode code point}
\]

### 5.2 Algorithm Conventions

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

Algorithms may be explicitly 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 \( \text{OperationName}(arg1, arg2) \). 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 \( \text{status} \) be SyntaxDirectedOperation of \( \text{SomeNonTerminal} \).
2. Let \( \text{someParseNode} \) be the parse of some source text.
3. Perform SyntaxDirectedOperation of \( \text{someParseNode} \).
4. Perform SyntaxDirectedOperation of \( \text{someParseNode} \) 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 reapply 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:

\[ \text{Block} : \{ \text{StatementList} \} \]

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**

\[ \text{Block} : \{ \text{StatementList} \} \]

1. Return the result of evaluating \( \text{StatementList} \).

### 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 \( \text{result} \) be \( \text{AbstractOperation}(\text{ReturnIfAbrupt}(\text{argument})) \).

mean the same thing as:

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

### 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. ? \( \text{OperationName}() \).

is equivalent to the following step:

1. \( \text{ReturnIfAbrupt}() \).

Similarly, for method application style, the step:

1. ? \( \text{someValue}.\text{OperationName}() \).

is equivalent to:

1. \( \text{ReturnIfAbrupt}() \).

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 \( \text{val} \) be ! \( \text{OperationName}() \).

is equivalent to the following steps:

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

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

1. Perform ! \( \text{SyntaxDirectedOperation of NonTerminal} \).

### 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 +∞ and -∞.
- **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 _u1D53D_ refers to Numbers, and the subscript _ℤ_ refers to BigInts. Numeric values without a subscript suffix refer to mathematical values.

Numeric operators such as +, ×, =, and ≥ 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 _ℝ_( _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 _ℤ_( _x_ ). A conversion from a Number or BigInt _x_ to a mathematical value is denoted as "the mathematical value of _x_", or _ℝ_( _x_ ). The mathematical value of +0_F and -0_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 +∞ and -∞ for +∞_F_ and -∞_F_ respectively; it is not defined for NaN.

The mathematical function abs(_x_) produces the absolute value of _x_, which is -_x_ if _x_ < 0 and otherwise is _x_ itself.
The mathematical function \( \text{min}(x_1, x_2, \ldots, x_N) \) produces the mathematically smallest of \( x_1 \) through \( x_N \). The mathematical function \( \text{max}(x_1, x_2, \ldots, x_N) \) produces the mathematically largest of \( x_1 \) through \( x_N \). The 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 \( \text{lower} \) and \( \text{upper} \)" (where \( x \) is an extended mathematical value and \( \text{lower} \) and \( \text{upper} \) are mathematical values such that \( \text{lower} \leq \text{upper} \)) produces \( \text{lower} \) if \( x < \text{lower} \), produces \( \text{upper} \) 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 \( \text{min} \), \( \text{max} \), \( \text{abs} \), and \( \text{floor} \) 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 \[ \text{floor}(x) = x - (x \mod 1). \]

5.2.6 Value Notation

In this specification, ECMAScript language values are displayed in bold. Examples include \texttt{null}, \texttt{true}, or "hello". These are distinguished from longer ECMAScript code sequences such as \texttt{Function.prototype.apply} or \texttt{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 “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 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 \(c_1\) is a leading surrogate and the second code unit \(c_2\) a trailing surrogate, is a surrogate pair and is interpreted as a code point with the value \((c_1 - 0xD800) \times 0x400 + (c_2 - 0xDC00) + 0x10000\). (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 \(A, B, \ldots\)" (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 $\text{inclusiveStart}$ to $\text{exclusiveEnd}$” (where $S$ is a String value or a sequence of code units and $\text{inclusiveStart}$ and $\text{exclusiveEnd}$ are integers) denotes the String value consisting of the consecutive code units of $S$ beginning at index $\text{inclusiveStart}$ and ending immediately before index $\text{exclusiveEnd}$ (which is the empty String when $\text{inclusiveStart} = \text{exclusiveEnd}$). If the “to” suffix is omitted, the length of $S$ is used as the value of $\text{exclusiveEnd}$.

### 6.1.4.1 StringIndexOf ( $\text{string, searchValue, fromIndex}$ )

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

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

**NOTE 1** If $\text{searchValue}$ is the empty String and $\text{fromIndex}$ is less than or equal to the length of $\text{string}$, this algorithm returns $\text{fromIndex}$. 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 $\text{fromIndex} >$ the length of $\text{string}$.

### 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 $[[\text{Description}]]$ that is either $\text{undefined}$ 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 $@@\text{name}$, where “name” is one of the values listed in Table 1.

<p>| Table 1: Well-known Symbols | 73 |</p>
<table>
<thead>
<tr>
<th>Specification Name</th>
<th>[[Description]]</th>
<th>Value and Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@asyncIterator</td>
<td>&quot;Symbol.asyncIterator&quot;</td>
<td>A method that returns the default AsyncIterator for an object. Called by the semantics of the <strong>for-await-of</strong> statement.</td>
</tr>
<tr>
<td>@@hasInstance</td>
<td>&quot;Symbol.hasInstance&quot;</td>
<td>A method that determines if a constructor object recognizes an object as one of the constructor's instances. Called by the semantics of the <strong>instanceof</strong> operator.</td>
</tr>
<tr>
<td>@@isConcatSpreadable</td>
<td>&quot;Symbol.isConcatSpreadable&quot;</td>
<td>A Boolean valued property that if true indicates that an object should be flattened to its array elements by <strong>Array.prototype.concat</strong>.</td>
</tr>
<tr>
<td>@@iterator</td>
<td>&quot;Symbol.iterator&quot;</td>
<td>A method that returns the default Iterator for an object. Called by the semantics of the <strong>for-of</strong> statement.</td>
</tr>
<tr>
<td>@@match</td>
<td>&quot;Symbol.match&quot;</td>
<td>A regular expression method that matches the regular expression against a string. Called by the <strong>String.prototype.match</strong> method.</td>
</tr>
<tr>
<td>@@matchAll</td>
<td>&quot;Symbol.matchAll&quot;</td>
<td>A regular expression method that returns an iterator, that yields matches of the regular expression against a string. Called by the <strong>String.prototype.matchAll</strong> method.</td>
</tr>
<tr>
<td>@@replace</td>
<td>&quot;Symbol.replace&quot;</td>
<td>A regular expression method that replaces matched substrings of a string. Called by the <strong>String.prototype.replace</strong> method.</td>
</tr>
<tr>
<td>@@search</td>
<td>&quot;Symbol.search&quot;</td>
<td>A regular expression method that returns the index within a string that matches the regular expression. Called by the <strong>String.prototype.search</strong> method.</td>
</tr>
<tr>
<td>@@species</td>
<td>&quot;Symbol.species&quot;</td>
<td>A function valued property that is the constructor function that is used to create derived objects.</td>
</tr>
<tr>
<td>@@split</td>
<td>&quot;Symbol.split&quot;</td>
<td>A regular expression method that splits a string at the indices that match the regular expression. Called by the <strong>String.prototype.split</strong> method.</td>
</tr>
<tr>
<td>@@toPrimitive</td>
<td>&quot;Symbol.toPrimitive&quot;</td>
<td>A method that converts an object to a corresponding primitive value. Called by the ToPrimitive abstract operation.</td>
</tr>
<tr>
<td>@@toStringTag</td>
<td>&quot;Symbol.toStringTag&quot;</td>
<td>A String valued property that is used in the creation of the default string description of an object. Accessed by the built-in method <strong>Object.prototype.toString</strong>.</td>
</tr>
<tr>
<td>@@unscopables</td>
<td>&quot;Symbol.unscopables&quot;</td>
<td>An object valued property whose own and inherited property names are property names that are excluded from the <strong>with</strong> environment bindings of the associated object.</td>
</tr>
</tbody>
</table>
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 $T$::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

<table>
<thead>
<tr>
<th>Invocation Synopsis</th>
<th>Example source</th>
<th>Invoked by the Evaluation semantics of ...</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$:unaryMinus(x)</td>
<td>$-x$</td>
<td>Unary $-$ Operator</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:bitwiseNOT(x)</td>
<td>$\sim x$</td>
<td>Bitwise NOT Operator ($\sim$)</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:exponentiate(x, y)</td>
<td>$x \text{ ** } y$</td>
<td>Exponentiation Operator and Math.pow (base, exponent)</td>
<td>$T$, may throw RangeError</td>
</tr>
<tr>
<td>$T$:multiply(x, y)</td>
<td>$x * y$</td>
<td>Multiplicative Operators</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:divide(x, y)</td>
<td>$x / y$</td>
<td>Multiplicative Operators</td>
<td>$T$, may throw RangeError</td>
</tr>
<tr>
<td>$T$:remainder(x, y)</td>
<td>$x % y$</td>
<td>Multiplicative Operators</td>
<td>$T$, may throw RangeError</td>
</tr>
<tr>
<td>$T$:add(x, y)</td>
<td>$x ++$ ++ $x$</td>
<td>Postfix Increment Operator, Prefix Increment Operator, and The Addition Operator ($+$)</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:subtract(x, y)</td>
<td>$x --$ -- $x$</td>
<td>Postfix Decrement Operator, Prefix Decrement Operator, and The Subtraction Operator ($-$)</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:leftShift(x, y)</td>
<td>$x &lt;&lt; y$</td>
<td>The Left Shift Operator ($&lt;&lt;$)</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:signedRightShift(x, y)</td>
<td>$x &gt;&gt; y$</td>
<td>The Signed Right Shift Operator ($&gt;&gt;$)</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$:unsignedRightShift(x, y)</td>
<td>$x &gt;&gt;&gt; y$</td>
<td>The Unsigned Right Shift Operator ($&gt;&gt;&gt;$)</td>
<td>$T$, may throw TypeError</td>
</tr>
<tr>
<td>$T$:lessThan(x, y)</td>
<td>$x &lt; y$ $x &gt; y$ $x \leq y$ $x \geq y$</td>
<td>Relational Operators, via Abstract Relational Comparison</td>
<td>Boolean or undefined (for unordered inputs)</td>
</tr>
<tr>
<td>$T$:equal(x, y)</td>
<td>$x == y$ $x != y$ $x === y$ $x !== y$</td>
<td>Equality Operators, via Strict Equality Comparison</td>
<td>Boolean</td>
</tr>
<tr>
<td>$T$:sameValue(x, y)</td>
<td></td>
<td>Object internal methods, via SameValue ($x, y$), to test</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
The `T::sameValueZero(x, y)` method is used by `Array`, `Map`, and `Set` methods, via `SameValueZero(x, y)`, to test value equality ignoring differences among members of the zero cohort (i.e., `-0_f` and `+0_f`).

### Binary Bitwise Operators

- `T::bitwiseAND(x, y)`
  ```
  x & y
  ```
  - `T` - Binary Bitwise Operators

- `T::bitwiseXOR(x, y)`
  ```
  x ^ y
  ```
  - `T` - Binary Bitwise Operators

- `T::bitwiseOR(x, y)`
  ```
  x | y
  ```
  - `T` - Binary Bitwise Operators

- `T::toString(x)`
  ```
  String(x)
  ```
  - Many expressions and built-in functions, via `ToString(argument)`
  - `String` - Value Type

---

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.

### 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 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.

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

---

**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.

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 `$+\infty_f$` and `$-\infty_f$`, 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 \(+0_F\) and \(-0_F\), respectively. (Note that these two different zero Number values are produced by the program expressions \(+0 +0\) (or simply \(0 0\)) and \(-0 -0\).)

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: \(+0_F\) and \(-0_F\).

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 \(\pi\)) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with \(-0_F\) 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 \(+\infty_F\); if \(-2^{1024}\) was chosen, replace it with \(-\infty_F\); if \(+0_F\) was chosen, replace it with \(-0_F\) 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 \(+\infty_F\), and the Number value for \(-\infty\) is \(-\infty_F\).

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_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 \(+0\) or \( exponent \) is \(-0\), return \( 1 \).
3. If \( base \) is NaN, return NaN.
4. If \( base \) is \( +\infty \), then
   a. If \( exponent > +0 \), return \( +\infty \). Otherwise, return \( +0 \).
5. If \( base \) is \( -\infty \), then
   a. If \( exponent > +0 \), then
      i. If \( exponent \) is an odd integral Number, return \( -\infty \). Otherwise, return \( +\infty \).
   b. Else,
      i. If \( exponent \) is an odd integral Number, return \( -0 \). Otherwise, return \( +0 \).
6. If \( base \) is \( +0 \), then
   a. If \( exponent > +0 \), return \( +0 \). Otherwise, return \( +\infty \).
7. If \( base \) is \( -0 \), then
   a. If \( exponent > +0 \), then
      i. If \( exponent \) is an odd integral Number, return \( -0 \). Otherwise, return \( +0 \).
   b. Else,
      i. If \( exponent \) is an odd integral Number, return \( -\infty \). Otherwise, return \( +\infty \).
8. Assert: \( base \) is finite and is neither \( +0 \) nor \( -0 \).
9. If \( exponent \) is \( +\infty \), then
   a. If \( \text{abs}(\mathbb{R}(base)) > 1 \), return \( +\infty \).
   b. If \( \text{abs}(\mathbb{R}(base)) = 1 \), return NaN.
   c. If \( \text{abs}(\mathbb{R}(base)) < 1 \), return \( +0 \).
10. If \( exponent \) is \( -\infty \), then
    a. If \( \text{abs}(\mathbb{R}(base)) > 1 \), return \( +0 \).
    b. If \( \text{abs}(\mathbb{R}(base)) = 1 \), return NaN.
    c. If \( \text{abs}(\mathbb{R}(base)) < 1 \), return \( +\infty \).
11. Assert: \( exponent \) is finite and is neither \( +0 \) nor \( -0 \).
12. If \( base < +0 \) 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 \( \mathbb{R}(exponent) \)
NOTE

The result of \( base \uparrow exponent \) when \( base \) is \( 1_F \) or \(-1_F \) and \( exponent \) is \(+\infty_F \) or \(-\infty_F \), or when \( base \) is \( 1_F \) 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 \( 1_F \). 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 \(+\infty_F \) or \( x \) is \(-\infty_F \), then
   a. If \( y \) is \(+0_F \) or \( y \) is \(-0_F \), return NaN.
   b. If \( y > +0_F \), return \( x \).
   c. Return \(-x\).
3. If \( y \) is \(+\infty_F \) or \( y \) is \(-\infty_F \), then
   a. If \( x \) is \(+0_F \) or \( x \) is \(-0_F \), return NaN.
   b. If \( x > +0_F \), return \( y \).
   c. Return \(-y\).
4. Return \( F(\mathbb{R}(x) \times \mathbb{R}(y)) \).

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 \(+\infty_F \) or \( x \) is \(-\infty_F \), then
   a. If \( y \) is \(+\infty_F \) or \( y \) is \(-\infty_F \), return NaN.
   b. If \( y > +0_F \), return \( x \).
   c. Return \(-x\).
3. If \( y \) is \(+\infty_F \), then
   a. If \( x \) is \(+0_F \) or \( x \) is \(+0_F \), return \( +0_F \). Otherwise, return \(-0_F \).
4. If \( y \) is \(-\infty_F \), then
   a. If \( x \) is \(+0_F \) or \( x \) is \(+0_F \), return \( -0_F \). Otherwise, return \(+0_F \).
5. If \( x \) is \(+0_F \) or \( x \) is \(-0_F \), then
   a. If \( y \) is \(+0_F \) or \( y \) is \(-0_F \), return NaN.
   b. If \( y > +0_F \), return \( x \).
   c. Return \(-x\).
6. If \( y \) is \(+0_F \), then
a. If \( x > +0_F \), return \(+\infty_F\). Otherwise, return \(-\infty_F\).

7. If \( y \) is \(-0_F\), then
   a. If \( x > +0_F \) return \(-\infty_F\). Otherwise, return \(+\infty_F\).

8. Return \( F(\mathbb{R}(x) / \mathbb{R}(y)) \).

### 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 \( \text{NaN} \) or \( d \) is \( \text{NaN} \), return \( \text{NaN} \).
2. If \( n \) is \( +\infty_F \) or \( n \) is \( -\infty_F \), return \( \text{NaN} \).
3. If \( d \) is \( +\infty_F \) or \( d \) is \( -\infty_F \), return \( n \).
4. If \( d \) is \( +0_F \) or \( d \) is \( -0_F \), return \( \text{NaN} \).
5. If \( n \) is \( +0_F \) or \( n \) is \( -0_F \), 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 \( F(r) \).

### 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 \( \text{NaN} \) or \( y \) is \( \text{NaN} \), return \( \text{NaN} \).
2. If \( x \) is \( +\infty_F \) and \( y \) is \( -\infty_F \), return \( \text{NaN} \).
3. If \( x \) is \( -\infty_F \) and \( y \) is \( +\infty_F \), return \( \text{NaN} \).
4. If \( x \) is \( +\infty_F \) or \( x \) is \( -\infty_F \), return \( x \).
5. If \( y \) is \( +\infty_F \) or \( y \) is \( -\infty_F \), return \( y \).
6. Assert: \( x \) and \( y \) are both finite.
7. If \( x \) is \( -0_F \) and \( y \) is \( -0_F \), return \(-0_F \).
8. Return \( F(\mathbb{R}(x) + \mathbb{R}(y)) \).
NOTE Finite-precision addition is commutative, but not always associative.

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)).

NOTE It is always the case that x - y produces the same result as x + (-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 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 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 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\(_F\) and \( y \) is -0\(_F\), return false.
5. If \( x \) is -0\(_F\) and \( y \) is +0\(_F\), return false.
6. If \( x \) is +∞\(_F\), return false.
7. If \( y \) is +∞\(_F\), return true.
8. If \( y \) is -∞\(_F\), return false.
9. If \( x \) is -∞\(_F\), 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\(_F\) and \( y \) is -0\(_F\), return true.
5. If \( x \) is -0\(_F\) and \( y \) is +0\(_F\), 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\(_F\) and \( y \) is -0\(_F\), return false.
3. If \( x \) is -0\(_F\) and \( y \) is +0\(_F\), 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\(_F\) and \( y \) is -0\(_F\), return true.
3. If \( x \) is -0\(_F\) and \( y \) is +0\(_F\), 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 \( \text{Inum} \) be \! ToInt32(\( x \)).  
3. Let \( \text{Rnum} \) be \! ToInt32(\( y \)).  
4. Let \( \text{Lbits} \) be the 32-bit two's complement bit string representing \( \mathbb{R}(\text{Inum}) \).  
5. Let \( \text{Rbits} \) be the 32-bit two's complement bit string representing \( \mathbb{R}(\text{Rnum}) \).  
6. If \( op \) is \&\&, let \( \text{result} \) be the result of applying the bitwise AND operation to \( \text{Lbits} \) and \( \text{Rbits} \).  
7. Else if \( op \) is ^\^, let \( \text{result} \) be the result of applying the bitwise exclusive OR (XOR) operation to \( \text{Lbits} \) and \( \text{Rbits} \).  
8. Else, \( op \) is |\|. Let \( \text{result} \) be the result of applying the bitwise inclusive OR operation to \( \text{Lbits} \) and \( \text{Rbits} \).  
9. Return the Number value for the integer represented by the 32-bit two's complement bit string \( \text{result} \).

**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\( \text{F} \) or -0\( \text{F} \), return the String "0".
3. If \( x \) < +0\( \text{F} \), return the string-concatenation of "-" and ! Number::toString(-\( x \)).
4. If \( x \) is +\( \text{∞F} \), 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:
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\(_u\), then \(\text{ToNumber(ToString}(x))\) is exactly the same **Number value** as \(x\).
- The least significant digit of \(s\) is not always uniquely determined by the requirements listed in step 5.

**NOTE 2**

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

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.
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 $x = 0_{\mathbb{Z}}$, 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, throw a RangeError exception.
2. Let quotient be \( \frac{\mathbb{R}(x)}{\mathbb{R}(y)} \).
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, throw a RangeError exception.
2. If n is 0, return 0.
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, then
   a. Return the BigInt value that represents \( \frac{\mathbb{R}(x)}{2^y} \), 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
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 \( \text{BinaryAnd}(x \text{ modulo } 2, y \text{ modulo } 2) \).
8. Else if \( op \) is | |, let \( tmp \) be \( \text{BinaryOr}(x \text{ modulo } 2, y \text{ modulo } 2) \).
9. Else,
   a. **Assert**: \( op \) is ^ ^.
   b. Let \( tmp \) be \( \text{BinaryXor}(x \text{ modulo } 2, y \text{ modulo } 2) \).
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 \( \text{BigIntBitwiseOp}(\&, x, y) \).

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 $\text{BigIntBitwiseOp}(\land, x, y)$.

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 $\text{BigIntBitwiseOp}(\lor, x, y)$.

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 $!\text{BigInt::toString}(-x)$.
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 $+0_{\mathbb{Z}}$ or a positive integral Number $\leq 2^{32} - 1$. An array index is an integer index whose numeric value $i$ is in the range $+0_{\mathbb{Z}} \leq i < 2^{32} - 1$.

Property keys are used to access properties and their values. There are two kinds of access for properties: get and set, corresponding to value retrieval and assignment, respectively. The properties accessible via get and set access includes both own properties that are a direct part of an object and inherited properties which are provided by another associated object via a property inheritance relationship. Inherited properties may be either own or inherited properties of the associated object. Each own 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>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>Any ECMAScript</td>
<td>The value retrieved by a get access of the property.</td>
</tr>
<tr>
<td></td>
<td>language type</td>
<td></td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>Boolean</td>
<td>If false, attempts by ECMAScript code to change the property’s [[Value]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>attribute using [[Set]] will not succeed.</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>Boolean</td>
<td>If true, the property will be enumerated by a for-in enumeration (see 14.7.5).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise, the property is said to be non-enumerable.</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>Boolean</td>
<td>If false, attempts to delete the property, change the property to be an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accessor property, or change its attributes (other than [[Value]], or changing [[Writable]] to false) will fail.</td>
</tr>
</tbody>
</table>

An accessor property associates a key value with the attributes listed in Table 4.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Get]]</td>
<td>Object</td>
<td>Undefined</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>Object</td>
<td>Undefined</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in Table 5 is used.
Table 5: Default Attribute Values

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>false</td>
</tr>
</tbody>
</table>

6.1.7.2 Object Internal Methods and Internal Slots

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

Internal method names are polymorphic. This means that different object values may perform different algorithms when a common internal method name is invoked upon them. 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

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[GetPrototypeOf]]</td>
<td>( ) → Object</td>
<td>Null</td>
</tr>
<tr>
<td>[[SetPrototypeOf]]</td>
<td>(Object</td>
<td>Null) → Boolean</td>
</tr>
<tr>
<td>[[IsExtensible]]</td>
<td>( ) → Boolean</td>
<td></td>
</tr>
<tr>
<td>[[PreventExtensions]]</td>
<td>( ) → Boolean</td>
<td></td>
</tr>
<tr>
<td>[[GetOwnProperty]]</td>
<td>(propertyKey) → Undefined</td>
<td></td>
</tr>
<tr>
<td>[[DefineOwnProperty]]</td>
<td>(propertyKey, PropertyDescriptor) → Boolean</td>
<td>Create or alter the own property, whose key is propertyKey, to have the state described by PropertyDescriptor. Return true if that property was successfully created/updated or false if the property could not be created or updated.</td>
</tr>
<tr>
<td>[[HasProperty]]</td>
<td>(propertyKey) → Boolean</td>
<td>Return a Boolean value indicating whether this object already has either an own or inherited property whose key is propertyKey.</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>(propertyKey, Receiver) → any</td>
<td>Return the value of the property whose key is propertyKey from this object. If any ECMAScript code must be executed to retrieve the property value, Receiver is used as the this value when evaluating the code.</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>(propertyKey, value, Receiver) → Boolean</td>
<td>Set the value of the property whose key is propertyKey to value. If any ECMAScript code must be executed to set the property value, Receiver is used as the this value when evaluating the code. Returns true if the property value was set or false if it could not be set.</td>
</tr>
<tr>
<td>[[Delete]]</td>
<td>(propertyKey) → Boolean</td>
<td>Remove the own property whose key is propertyKey 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.</td>
</tr>
<tr>
<td>[[OwnPropertyKeys]]</td>
<td>( ) → List of propertyKey</td>
<td>Return a List whose elements are all of the own property keys for the object.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Call]]</td>
<td>(any, a List of any) → any</td>
<td>Executes code associated with 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 callable.</td>
</tr>
<tr>
<td>[[Construct]]</td>
<td>(a List of any, Object) → Object</td>
<td>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 constructors. A function object is not necessarily a constructor and such non-constructor function objects do not have a [[Construct]] internal method.</td>
</tr>
</tbody>
</table>

The semantics of the essential internal 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 [[Extensible]] 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>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Global Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%AggregateError%</td>
<td>AggregateError</td>
<td>The AggregateError constructor (20.5.7.1)</td>
</tr>
<tr>
<td>%Array%</td>
<td>Array</td>
<td>The Array constructor (23.1.1)</td>
</tr>
<tr>
<td>%ArrayBuffer%</td>
<td>ArrayBuffer</td>
<td>The ArrayBuffer constructor (25.1.3)</td>
</tr>
<tr>
<td>%ArrayIteratorPrototype%</td>
<td></td>
<td>The prototype of Array iterator objects (23.1.5)</td>
</tr>
<tr>
<td>%AsyncFromSyncIteratorPrototype%</td>
<td></td>
<td>The prototype of async-from-sync iterator objects (27.1.4)</td>
</tr>
<tr>
<td>%AsyncFunction%</td>
<td></td>
<td>The constructor of async function objects (27.7.1)</td>
</tr>
<tr>
<td>%AsyncGeneratorFunction%</td>
<td></td>
<td>The constructor of async iterator objects (27.4.1)</td>
</tr>
<tr>
<td>%AsyncIteratorPrototype%</td>
<td></td>
<td>An object that all standard built-in async iterator objects indirectly inherit from</td>
</tr>
<tr>
<td>%Atomics%</td>
<td>Atomics</td>
<td>The Atomics object (25.4)</td>
</tr>
<tr>
<td>%BigInt%</td>
<td>BigInt</td>
<td>The BigInt constructor (21.2.1)</td>
</tr>
<tr>
<td>%BigInt64Array%</td>
<td>BigInt64Array</td>
<td>The BigInt64Array constructor (23.2)</td>
</tr>
<tr>
<td>%BigUint64Array%</td>
<td>BigInt64Array</td>
<td>The BigUint64Array constructor (23.2)</td>
</tr>
<tr>
<td>%Boolean%</td>
<td>Boolean</td>
<td>The Boolean constructor (20.3.1)</td>
</tr>
<tr>
<td>%DataView%</td>
<td>DataView</td>
<td>The DataView constructor (25.3.2)</td>
</tr>
<tr>
<td>%Date%</td>
<td>Date</td>
<td>The Date constructor (21.4.2)</td>
</tr>
<tr>
<td>%decodeURI%</td>
<td>decodeURI</td>
<td>The decodeURI function (19.2.6.2)</td>
</tr>
<tr>
<td>%decodeURIComponent%</td>
<td>decodeURIComponent</td>
<td>The decodeURIComponent function (19.2.6.3)</td>
</tr>
<tr>
<td>%encodeURI%</td>
<td>encodeURI</td>
<td>The encodeURI function (19.2.6.4)</td>
</tr>
<tr>
<td>%encodeURIComponent%</td>
<td>encodeURIComponent</td>
<td>The encodeURIComponent function (19.2.6.5)</td>
</tr>
<tr>
<td>%Error%</td>
<td>Error</td>
<td>The Error constructor (20.5.1)</td>
</tr>
<tr>
<td>%eval%</td>
<td>eval</td>
<td>The eval function (19.2.1)</td>
</tr>
<tr>
<td>%EvalError%</td>
<td>EvalError</td>
<td>The EvalError constructor (20.5.5.1)</td>
</tr>
<tr>
<td>%FinalizationRegistry%</td>
<td>FinalizationRegistry</td>
<td>The FinalizationRegistry constructor (26.2.1)</td>
</tr>
<tr>
<td>%Float32Array%</td>
<td>Float32Array</td>
<td>The Float32Array constructor (23.2)</td>
</tr>
<tr>
<td>%Float64Array%</td>
<td>Float64Array</td>
<td>The Float64Array constructor (23.2)</td>
</tr>
<tr>
<td>%ForInIteratorPrototype%</td>
<td></td>
<td>The prototype of For-In iterator objects (14.7.5.10)</td>
</tr>
<tr>
<td>%Function%</td>
<td>Function</td>
<td>The Function constructor (20.2.1)</td>
</tr>
<tr>
<td>%GeneratorFunction%</td>
<td></td>
<td>The constructor of generator objects (27.3.1)</td>
</tr>
<tr>
<td>%Int8Array%</td>
<td>Int8Array</td>
<td>The Int8Array constructor (23.2)</td>
</tr>
<tr>
<td>%Int16Array%</td>
<td>Int16Array</td>
<td>The Int16Array constructor (23.2)</td>
</tr>
<tr>
<td>%Int32Array%</td>
<td>Int32Array</td>
<td>The Int32Array constructor (23.2)</td>
</tr>
<tr>
<td>%isFinite%</td>
<td>isFinite</td>
<td>The isFinite function (19.2.2)</td>
</tr>
<tr>
<td>%isNaN%</td>
<td>isNaN</td>
<td>The isNaN function (19.2.3)</td>
</tr>
<tr>
<td>%IteratorPrototype%</td>
<td></td>
<td>An object that all standard built-in iterator objects indirectly inherit from</td>
</tr>
<tr>
<td>%JSON%</td>
<td>JSON</td>
<td>The JSON object (25.5)</td>
</tr>
<tr>
<td>%Map%</td>
<td>Map</td>
<td>The Map constructor (24.1.1)</td>
</tr>
<tr>
<td>%MapIteratorPrototype%</td>
<td></td>
<td>The prototype of Map iterator objects (24.1.5)</td>
</tr>
<tr>
<td>%Math%</td>
<td>Math</td>
<td>The <strong>Math</strong> object (21.3)</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>%Number%</td>
<td>Number</td>
<td>The Number constructor (21.1.1)</td>
</tr>
<tr>
<td>%Object%</td>
<td>Object</td>
<td>The Object constructor (20.1.1)</td>
</tr>
<tr>
<td>%parseFloat%</td>
<td>parseFloat</td>
<td>The <strong>parseFloat</strong> function (19.2.4)</td>
</tr>
<tr>
<td>%parseInt%</td>
<td>parseInt</td>
<td>The <strong>parseInt</strong> function (19.2.5)</td>
</tr>
<tr>
<td>%Promise%</td>
<td>Promise</td>
<td>The Promise constructor (27.2.3)</td>
</tr>
<tr>
<td>%Proxy%</td>
<td>Proxy</td>
<td>The Proxy constructor (28.2.1)</td>
</tr>
<tr>
<td>%RangeError%</td>
<td>RangeError</td>
<td>The RangeError constructor (20.5.5.2)</td>
</tr>
<tr>
<td>%ReferenceError%</td>
<td>ReferenceError</td>
<td>The ReferenceError constructor (20.5.5.3)</td>
</tr>
<tr>
<td>%Reflect%</td>
<td>Reflect</td>
<td>The <strong>Reflect</strong> object (28.1)</td>
</tr>
<tr>
<td>%RegExp%</td>
<td>RegExp</td>
<td>The RegExp constructor (22.2.3)</td>
</tr>
<tr>
<td>%RegExpStringIteratorPrototype%</td>
<td></td>
<td>The prototype of RegExp String Iterator objects (22.2.7)</td>
</tr>
<tr>
<td>%Set%</td>
<td>Set</td>
<td>The Set constructor (24.2.1)</td>
</tr>
<tr>
<td>%SetIteratorPrototype%</td>
<td></td>
<td>The prototype of Set iterator objects (24.2.5)</td>
</tr>
<tr>
<td>%SharedArrayBuffer%</td>
<td>SharedArrayBuffer</td>
<td>The SharedArrayBuffer constructor (25.2.2)</td>
</tr>
<tr>
<td>%String%</td>
<td>String</td>
<td>The String constructor (22.1.1)</td>
</tr>
<tr>
<td>%StringIteratorPrototype%</td>
<td></td>
<td>The prototype of String iterator objects (22.1.5)</td>
</tr>
<tr>
<td>%Symbol%</td>
<td>Symbol</td>
<td>The Symbol constructor (20.4.1)</td>
</tr>
<tr>
<td>%SyntaxError%</td>
<td>SyntaxError</td>
<td>The SyntaxError constructor (20.5.5.4)</td>
</tr>
<tr>
<td>%ThrowTypeError%</td>
<td></td>
<td>A function object that unconditionally throws a new instance of %TypeError%</td>
</tr>
<tr>
<td>%TypedArray%</td>
<td>TypeError</td>
<td>The super class of all typed Array constructors (23.2.1)</td>
</tr>
<tr>
<td>%TypedError%</td>
<td>TypeError</td>
<td>The TypeError constructor (20.5.5.5)</td>
</tr>
<tr>
<td>%Uint8Array%</td>
<td>Uint8Array</td>
<td>The Uint8Array constructor (23.2)</td>
</tr>
<tr>
<td>%Uint8ClampedArray%</td>
<td>Uint8ClampedArray</td>
<td>The Uint8ClampedArray constructor (23.2)</td>
</tr>
<tr>
<td>%Uint16Array%</td>
<td>Uint16Array</td>
<td>The Uint16Array constructor (23.2)</td>
</tr>
<tr>
<td>%Uint32Array%</td>
<td>Uint32Array</td>
<td>The Uint32Array constructor (23.2)</td>
</tr>
<tr>
<td>%URLError%</td>
<td>URIError</td>
<td>The URIError constructor (20.5.5.6)</td>
</tr>
</tbody>
</table>
6.2 ECMAScript Specification Types

A specification type corresponds to meta-values that are used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The specification types 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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Type]]</td>
<td>One of normal, break, continue, return, or throw</td>
<td>The type of completion that occurred.</td>
</tr>
<tr>
<td>[[Value]]</td>
<td>any ECMAScript language value or empty</td>
<td>The value that was produced.</td>
</tr>
<tr>
<td>[[Target]]</td>
<td>any ECMAScript string or empty</td>
<td>The target label for directed control transfers.</td>
</tr>
</tbody>
</table>
The term “abrupt completion” refers to any completion with a [[Type]] value other than normal.

### 6.2.3.1 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.
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.
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`. 

### 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.
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`. 

### 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 \( \text{Completion} \left( [\text{Type}]: \text{return}, [\text{Value}]: \text{argument}, [\text{Target}]: \text{empty} \right) \).

6.2.3.4 UpdateEmpty ( \( \text{completionRecord}, \text{value} \) )

The abstract operation UpdateEmpty takes arguments \( \text{completionRecord} \) and \( \text{value} \). It performs the following steps when called:

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

6.2.4 The Reference Record Specification Type

The \textit{Reference Record} type is used to explain the behaviour of such operators as \textit{delete}, \textit{typeof}, the assignment operators, the \textit{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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Base]]</td>
<td>One of:</td>
<td>The value or \textit{Environment Record} which holds the binding. A [[Base]] of \text{unresolvable} indicates that the binding could not be resolved.</td>
</tr>
<tr>
<td></td>
<td>• any \textit{ECMAScript}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>language value except undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or \text{null},</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• an \textit{Environment}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Record, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• \text{unresolvable}.</td>
<td></td>
</tr>
<tr>
<td>[[ReferencedName]]</td>
<td>String or Symbol</td>
<td>The name of the binding. Always a String if [[Base]] value is an \textit{Environment Record}.</td>
</tr>
<tr>
<td>[[Strict]]</td>
<td>Boolean</td>
<td>true if the \textit{Reference Record} originated in \textit{strict mode code}, false otherwise.</td>
</tr>
<tr>
<td>[[ThisValue]]</td>
<td>any \textit{ECMAScript}</td>
<td>If not \text{empty}, the \textit{Reference Record} represents a property binding that was expressed using the \textit{super} keyword; it is called a \textit{Super Reference Record} and its [[Base]] value will never be an \textit{Environment Record}. In that case, the [[ThisValue]] field holds the this value at the time the \textit{Reference Record} was created.</td>
</tr>
</tbody>
</table>
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. ReturnIfAbrupt(W).
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 \( V.\\text{[[Strict]]} \) is true, throw a ReferenceError exception.
   b. Let globalObj be GetGlobalObject().
   c. Return ? Set(globalObj, \( V.\\text{[[ReferencedName]]} \), \( W \), false).

5. If IsPropertyReference(\( V \)) is true, then
   a. Let baseObj be ! ToObject(\( V.\\text{[[Base]]} \)).
   b. Let succeeded be ? baseObj.\\text{[[Set]]}(\( V.\\text{[[ReferencedName]]} \), \( W \), GetThisValue(\( V \))).
   c. If succeeded is false and \( V.\\text{[[Strict]]} \) is true, throw a TypeError exception.
   d. Return.

6. Else,
   a. Let base be \( V.\\text{[[Base]]} \).
   b. Assert: base is an Environment Record.
   c. Return ? base.SetMutableBinding(\( V.\\text{[[ReferencedName]]} \), \( W \), \( V.\\text{[[Strict]]} \)) (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 \( V.\\text{[[ThisValue]]} \); otherwise return \( V.\\text{[[Base]]} \).

### 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 \( V.\\text{[[Base]]} \).
6. Assert: base is an Environment Record.
7. Return base.InitializeBinding(\( V.\\text{[[ReferencedName]]} \), \( W \)).

### 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 + 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-origined 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 ( \( \text{size} \) )

The abstract operation CreateByteDataBlock takes argument \( \text{size} \) (an integer). It performs the following steps when called:

1. Assert: \( \text{size} \geq 0 \).
2. Let \( db \) be a new Data Block value consisting of \( \text{size} \) 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 ( \( \text{size} \) )

The abstract operation CreateSharedByteDataBlock takes argument \( \text{size} \) (a non-negative integer). It performs the following steps when called:

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

### 6.2.8.3 CopyDataBlockBytes ( \( \text{toBlock}, \text{toIndex}, \text{fromBlock}, \text{fromIndex}, \text{count} \) )

The abstract operation CopyDataBlockBytes takes arguments \( \text{toBlock}, \text{toIndex} \) (a non-negative integer), \( \text{fromBlock}, \text{fromIndex}, \text{count} \) (a non-negative integer). It performs the following steps when called:

1. Assert: \( \text{size} \geq 0 \).
2. Let \( db \) be a new Data Block value consisting of \( \text{size} \) 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 \( db \).
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]].
   b. Else, 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.
   c. Else, set toBlock[toIndex] to bytes[0].
   d. Otherwise, set toBlock[toIndex] to fromBlock[fromIndex].
   e. Set toIndex to toIndex + 1.
5. Set fromIndex to fromIndex + 1.
6. 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>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Return <code>false</code>.</td>
</tr>
<tr>
<td>Null</td>
<td>Return <code>false</code>.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return <code>argument</code>.</td>
</tr>
<tr>
<td>Number</td>
<td>If <code>argument</code> is <code>+0</code>, <code>-0</code>, or <code>NaN</code>, return <code>false</code>; otherwise return <code>true</code>.</td>
</tr>
<tr>
<td>String</td>
<td>If <code>argument</code> is the empty String (its length is 0), return <code>false</code>; otherwise return <code>true</code>.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return <code>true</code>.</td>
</tr>
<tr>
<td>BigInt</td>
<td>If <code>argument</code> is <code>0</code>, return <code>false</code>; otherwise return <code>true</code>.</td>
</tr>
<tr>
<td>Object</td>
<td>Return <code>true</code>.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Return NaN.</td>
</tr>
<tr>
<td>Null</td>
<td>Return +0.</td>
</tr>
<tr>
<td>Boolean</td>
<td>If <code>argument</code> is true, return 1. If <code>argument</code> is false, return +0.</td>
</tr>
<tr>
<td>Number</td>
<td>Return <code>argument</code> (no conversion).</td>
</tr>
<tr>
<td>String</td>
<td>See grammar and conversion algorithm below.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Throw a <strong>TypeError</strong> exception.</td>
</tr>
<tr>
<td>BigInt</td>
<td>Throw a <strong>TypeError</strong> exception.</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1. Let <code>primValue</code> be ? ToPrimitive(<code>argument</code>, number).</td>
</tr>
<tr>
<td></td>
<td>2. Return ? ToNumber(<code>primValue</code>).</td>
</tr>
</tbody>
</table>

### 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** :::
  - `StrWhiteSpace`<sub>opt</sub>
  - `StrWhiteSpace`<sub>opt</sub> `StrNumericLiteral` `StrWhiteSpace`<sub>opt</sub>

- **StrWhiteSpace** :::
  - `StrWhiteSpaceChar` `StrWhiteSpace`<sub>opt</sub>

- **StrWhiteSpaceChar** :::
  - `WhiteSpace`
  - `LineTerminator`

- **StrNumericLiteral** :::
  - `StrDecimalLiteral`
  - `NonDecimalIntegerLiteral`<~Sep~>
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 +0F.
- `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 StringNumericLiteral StrWhiteSpace_opt` is the MV of `StringNumericLiteral`, 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 +0F or -0F 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 \(\text{StrUnsignedDecimalLiteral} ::: \text{DecimalDigits} \cdot \text{ExponentPart}\) is the MV of \(\text{DecimalDigits}\) times \(10^e\), where \(e\) is the MV of \(\text{ExponentPart}\).

The MV of \(\text{StrUnsignedDecimalLiteral} ::: \text{DecimalDigits} \cdot \text{DecimalDigits} \cdot \text{ExponentPart}\) is (the MV of the first \(\text{DecimalDigits}\) plus (the MV of the second \(\text{DecimalDigits}\) times \(10^n\))) times \(10^e\), where \(n\) is the number of code points in the second \(\text{DecimalDigits}\) and \(e\) is the MV of \(\text{ExponentPart}\).

The MV of \(\text{StrUnsignedDecimalLiteral} ::: \text{DecimalDigits} \cdot \text{DecimalDigits}\) is the MV of \(\text{DecimalDigits}\) times \(10^{-n}\), where \(n\) is the number of code points in \(\text{DecimalDigits}\).

The MV of \(\text{StrUnsignedDecimalLiteral} ::: \text{DecimalDigits} \cdot \text{ExponentPart}\) is the MV of \(\text{DecimalDigits}\) times \(10^{e-n}\), where \(n\) is the number of code points in \(\text{DecimalDigits}\) and \(e\) is the MV of \(\text{ExponentPart}\).

The MV of \(\text{StrUnsignedDecimalLiteral} ::: \text{ExponentPart}\) is the MV of \(\text{ExponentPart}\).

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 \(+0\) unless the first non white space code point in the String numeric literal is -, in which case the rounded value is \(-0\). Otherwise, the rounded value must be the Number value for the MV (in the sense defined in 6.1.6.1), unless the literal includes a \(\text{StrUnsignedDecimalLiteral}\) 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 \(\text{argument}\). It converts \(\text{argument}\) to an integer, \(+\infty\), or \(-\infty\). It performs the following steps when called:

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

### 7.1.6 ToInt32 ( argument )

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

1. Let \(\text{number}\) be \(\text{? ToNumber(\text{argument})}\).
2. If \(\text{number}\) is NaN, \(+0\), \(-0\), \(+\infty\), or \(-\infty\), return \(+0\).
3. Let \(\text{int}\) be the mathematical value that is the same sign as \(\text{number}\) and whose magnitude is floor(abs(\(\text{number}\)))).
4. Let \(\text{int32bit}\) be \(\text{int}\) modulo \(2^{32}\).
5. If \( \text{int32bit} \geq 2^{31} \), return \( \frac{\text{int32bit}}{2^{32}} \); otherwise return \( \frac{\text{int32bit}}{2^{32}} \).

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(ToInt32(\( x \))) is the same value as ToInt32(\( x \)) for all values of \( x \). (It is to preserve this latter property that +\( \infty \) and -\( \infty \) are mapped to +0.)
- ToInt32 maps -0 to +0.

7.1.7 ToUint32 ( \texttt{argument} )

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

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

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 +\( \infty \) and -\( \infty \) are mapped to +0.)
- ToUint32 maps -0 to +0.

7.1.8 ToInt16 ( \texttt{argument} )

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

1. Let \( \text{number} \) be \( ? \text{ToNumber}(\text{argument}) \).
2. If \( \text{number} \) is NaN, +0, -0, +\( \infty \), or -\( \infty \), return +0.
3. Let \( \text{int} \) be the mathematical value that is the same sign as \( \text{number} \) and whose magnitude is \( \text{floor}(\text{abs}(\Re(\text{number}))) \).
4. Let \( \text{int16bit} \) be \( \text{int} \) modulo \( 2^{16} \).
5. If \( \text{int16bit} \geq 2^{15} \), return \( \frac{\text{int16bit}}{2^{16}} \); otherwise return \( \frac{\text{int16bit}}{2^{16}} \).
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 $+0_F$ 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_F, -0_F, +\infty_F, or -\infty_F, return +0_F.
3. Let int be the mathematical value that is the same sign as number and whose magnitude is \(\left\lfloor \text{abs}(\mathbb{R}(number)) \right\rfloor\).
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_F to +0_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 $-128_F$ through $127_F$, inclusive. It performs the following steps when called:

1. Let number be ? ToNumber(argument).
2. If number is NaN, +0_F, -0_F, +\infty_F, or -\infty_F, return +0_F.
3. Let int be the mathematical value that is the same sign as number and whose magnitude is \(\left\lfloor \text{abs}(\mathbb{R}(number)) \right\rfloor\).
4. Let int8bit be int modulo $2^{8}$.
5. If int8bit ≥ $2^{7}$, 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 $+0_F$ through $255_F$, inclusive. It performs the following steps when called:

1. Let number be ? ToNumber(argument).
2. If number is NaN, +0_F, -0_F, +\infty_F, or -\infty_F, return +0_F.
3. Let int be the mathematical value that is the same sign as number and whose magnitude is \(\left\lfloor \text{abs}(\mathbb{R}(number)) \right\rfloor\).
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 $+0_F$ through $255_F$, inclusive. It performs the following steps when called:

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

NOTE Unlike the other ECMAScript integer conversion abstract operation, ToUint8Clamp rounds rather than truncates non-integral values and does not convert $+\infty_F$ to $+0_F$. 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>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return 1n if $prim$ is true and 0n if $prim$ is false.</td>
</tr>
<tr>
<td>BigInt</td>
<td>Return $prim$.</td>
</tr>
<tr>
<td>Number</td>
<td>Throw a TypeError exception.</td>
</tr>
</tbody>
</table>
| String        | 1. Let $n$ be ! StringToBigInt($prim$).
2. If $n$ is NaN, throw a SyntaxError exception.
3. Return $n$.                                |
| 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 $\text{int64bit} = R(n)$ 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 $\text{int64bit} = R(n)$ 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

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Return &quot;undefined&quot;.</td>
</tr>
<tr>
<td>Null</td>
<td>Return &quot;null&quot;.</td>
</tr>
<tr>
<td>Boolean</td>
<td>If argument is true, return &quot;true&quot;.</td>
</tr>
<tr>
<td></td>
<td>If argument is false, return &quot;false&quot;.</td>
</tr>
<tr>
<td>Number</td>
<td>Return ! Number::toString(argument).</td>
</tr>
<tr>
<td>String</td>
<td>Return argument.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>BigInt</td>
<td>Return ! BigInt::toString(argument).</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1. Let primValue be ? ToPrimitive(argument, string).</td>
</tr>
</tbody>
</table>

---

### 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

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

#### 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 +0𝔽.
3. Return 𝔽(min(`len`, 2^{53} - 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(\textit{argument}) is String.
2. If \textit{argument} is "-0", return -0\_\textup{e}.
3. Let \( n \) be \!\text{ToNumber} (\textit{argument}).
4. If \text{SameValue}(\!\text{ToString}(n), \textit{argument}) is \text{false}, return \text{undefined}.
5. Return \( n \).

A canonical numeric string is any String value for which the CanonicalNumericIndexString abstract operation does not return \text{undefined}.

### 7.1.22 ToIndex ( \textit{value} )

The abstract operation ToIndex takes argument \textit{value}. It returns \textit{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 \textit{value} is \text{undefined}, then
   a. Return 0.
2. Else,
   a. Let \textit{integerIndex} be \( \frac{1}{0} \text{ToIntegerOrInfinity} (\textit{value}) \).
   b. If \textit{integerIndex} < +0\_\textup{e}, throw a \textbf{RangeError} exception.
   c. Let \textit{index} be !\text{ToLength}(\textit{integerIndex}).
   d. If !\text{SameValue}(\textit{integerIndex}, \textit{index}) is \text{false}, throw a \textbf{RangeError} exception.
   e. Return \( \mathbb{R}(\textit{index}) \).

### 7.2 Testing and Comparison Operations

#### 7.2.1 RequireObjectCoercible ( \textit{argument} )

The abstract operation RequireObjectCoercible takes argument \textit{argument}. It throws an error if \textit{argument} is a value that cannot be converted to an Object using ToObject. It is defined by Table 16:

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a \textbf{TypeError} exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a \textbf{TypeError} exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return \textit{argument}.</td>
</tr>
<tr>
<td>Number</td>
<td>Return \textit{argument}.</td>
</tr>
<tr>
<td>String</td>
<td>Return \textit{argument}.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return \textit{argument}.</td>
</tr>
<tr>
<td>BigInt</td>
<td>Return \textit{argument}.</td>
</tr>
<tr>
<td>Object</td>
<td>Return \textit{argument}.</td>
</tr>
</tbody>
</table>
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]].
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, +∞, or −∞, return false.
3. If floor(abs(R(argument))) ≠ abs(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 +0\(_F\) and -0\(_F\).

### 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 ±∞ or \( ny \) is ±∞, return true.
      j. If \( nx \) is +∞ or \( ny \) is −∞, 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 \( x \equiv ! \text{ToNumber}(y) \).
6. If Type(x) is String and Type(y) is Number, return the result of the comparison \( ! \text{ToNumber}(x) \equiv y \).
7. If Type(x) is BigInt and Type(y) is String, then
   a. Let \( n \) be \( \text{StringToBigInt}(y) \).
   b. If \( n \) is NaN, return false.
   c. Return the result of the comparison \( x \equiv n \).
8. If Type(x) is String and Type(y) is BigInt, return the result of the comparison \( \text{ToNumber}(x) == y \).
9. If Type(x) is either String, Number, BigInt, or Symbol and Type(y) is Object, return the result of the comparison \( x \equiv ? \text{ToPrimitive}(y) \).
10. If Type(x) is Object and Type(y) is either String, Number, BigInt, or Symbol, return the result of the comparison \( ? \text{ToPrimitive}(x) == y \).
11. 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, +\( \infty \), or -\( \infty \), return false.
    b. If \( \mathbb{R}(x) = \mathbb{R}(y) \), return true; otherwise return false.
12. Return false.

### 7.2.15 Strict Equality Comparison

The comparison \( x \equiv y \), 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 \( ! \text{Type}(x)::_\text{equal}(x, y) \).
3. Return \( ! \text{SameValueNonNumeric}(x, y) \).

#### 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 \( \text{obj} \) 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.

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).

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.
5. If success is false and Throw is true, throw a TypeError exception.

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

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, [[DefineOwnProperty]] will return \texttt{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 \texttt{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 \texttt{true}.
3. Let \( \texttt{success} \) be ? CreateDataProperty\((O, P, V)\).
4. If \( \texttt{success} \) is \texttt{false}, throw a \texttt{TypeError} exception.
5. Return \( \texttt{success} \).

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, [[DefineOwnProperty]] will return \texttt{false} causing this operation to throw a \texttt{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 [[DefineOwnProperty]] internal method of an object in a manner that will throw a \texttt{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.
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.
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.

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.
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 \( \text{argumentsList} \) (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 \( \text{argumentsList} \) is the value passed to the corresponding argument of the internal method. If \( \text{argumentsList} \) is not present, a new empty List is used as its value. It performs the following steps when called:

1. If \( \text{argumentsList} \) is not present, set \( \text{argumentsList} \) to a new empty List.
2. If IsCallable(\( F \)) is false, throw a TypeError exception.

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

The abstract operation Construct takes argument \( F \) (a function object) and optional arguments \( \text{argumentsList} \) and newTarget. It is used to call the [[Construct]] internal method of a function object. \( \text{argumentsList} \) and newTarget are the values to be passed as the corresponding arguments of the internal method. If \( \text{argumentsList} \) 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 \( \text{argumentsList} \) is not present, set \( \text{argumentsList} \) 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(\ldots\text{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 $\mathbb{R}(\operatorname{ToLength}(\operatorname{Get}(obj, "length")))$.

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 \([\text{BoundTargetFunction}]\) internal slot, then
   a. Let \( BC \) be \( C.[\text{BoundTargetFunction}] \).
   b. Return ? InstanceofOperator(\( O, BC \)).
3. If Type(\( O \)) is not Object, return false.
4. Let \( P \) be ? Get(\( C, "\text{prototype}" \)).
5. If Type(\( P \)) is not Object, throw a TypeError exception.
6. Repeat,
   a. Set \( O \) to ? \( O.[\text{GetPrototypeOf}]() \).
   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(\( O, "\text{constructor}" \)).
3. If \( C \) is undefined, return defaultConstructor.
4. If Type(\( C \)) is not Object, throw a TypeError exception.
5. Let \( S \) be ? Get(\( C, @@\text{species} \)).
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 \( O.[\text{OwnPropertyKeys}]() \).
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 \( O.[\text{GetOwnProperty}](\text{key}) \).
      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`.
### 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.
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 \( \text{return} \) is **undefined**, return \( \text{Completion}(\text{completion}) \).

c. Set \( \text{innerResult} \) to \( \text{Call}(\text{return}, \text{iterator}) \).

d. If \( \text{innerResult}.[[\text{Type}]] \) is **normal**, set \( \text{innerResult} \) to \( \text{Await}(\text{innerResult}.[[\text{Value}]]) \).

6. If \( \text{completion}.[[\text{Type}]] \) is **throw**, return \( \text{Completion}(\text{completion}) \).

7. If \( \text{innerResult}.[[\text{Type}]] \) is **throw**, return \( \text{Completion}(\text{innerResult}) \).

8. If \( \text{Type}(\text{innerResult}.[[\text{Value}]]) \) is not **Object**, throw a **TypeError** exception.

9. Return \( \text{Completion}(\text{completion}) \).

### 7.4.8 CreateIterResultObject ( value, done )

The abstract operation CreateIterResultObject takes arguments \( \text{value} \) and \( \text{done} \). It creates an object that supports the IteratorResult interface. It performs the following steps when called:

1. **Assert**: Type(\( \text{done} \)) is Boolean.

2. Let \( \text{obj} \) be \( \text{! OrdinaryObjectCreate}(%\text{Object.prototype}%) \).

3. Perform \( \text{! CreateDataPropertyOrThrow}(%\text{obj}%, "\text{value}", \text{value}) \).

4. Perform \( \text{! CreateDataPropertyOrThrow}(%\text{obj}%, "\text{done}", \text{done}) \).

5. Return \( \text{obj} \).

### 7.4.9 CreateListIteratorRecord ( list )

The abstract operation CreateListIteratorRecord takes argument \( \text{list} \). It creates an Iterator (27.1.1.2) object record whose next method returns the successive elements of \( \text{list} \). It performs the following steps when called:

1. Let \( \text{closure} \) be a new Abstract Closure with no parameters that captures \( \text{list} \) and performs the following steps when called:
   a. For each element \( \text{E} \) of \( \text{list} \), do
      i. Perform ? \( \text{Yield}(\text{E}) \).
   b. Return **undefined**.

2. Let \( \text{iterator} \) be \( \text{! CreateIteratorFromClosure}(%\text{closure}%, \text{empty}, %\text{IteratorPrototype}%) \).

3. Return \( \text{Record} \{ [[\text{Iterator}]]: \text{iterator}, [[\text{NextMethod}]]: %\text{GeneratorFunction.prototype.prototype.next}%, [[\text{Done}]]: \text{false} \} \).

**NOTE** The list iterator object is never directly accessible to ECMAScript code.

### 7.4.10 IterableToList ( items [ , method ] )

The abstract operation IterableToList takes argument \( \text{items} \) and optional argument \( \text{method} \). It performs the following steps when called:

1. If \( \text{method} \) is present, then
   a. Let \( \text{iteratorRecord} \) be ? GetIterator(\( \text{items} \), \( \text{sync} \), \( \text{method} \)).

2. Else,
   a. Let \( \text{iteratorRecord} \) be ? GetIterator(\( \text{items} \), \( \text{sync} \)).

3. Let \( \text{values} \) be a new empty \( \text{List} \).

4. Let \( \text{next} \) be **true**.

5. Repeat, while \( \text{next} \) 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 \( \textit{names} \) be \( \text{BoundNames} \) of \( \text{VariableDeclarationList} \).
2. Append to \( \textit{names} \) the elements of \( \text{BoundNames} \) of \( \text{VariableDeclaration} \).
3. Return \( \textit{names} \).

**VariableDeclaration** : \( \text{BindingIdentifier} \ \text{Initializer}_{\text{opt}} \)

1. Return the \( \text{BoundNames} \) of \( \text{BindingIdentifier} \).

**VariableDeclaration** : \( \text{BindingPattern} \ \text{Initializer} \)

1. Return the \( \text{BoundNames} \) of \( \text{BindingPattern} \).

**ObjectBindingPattern** : \{ \}

1. Return a new empty \( \text{List} \).

**ObjectBindingPattern** : \{ \text{BindingPropertyList} , \text{BindingRestProperty} \}

1. Let \( \textit{names} \) be \( \text{BoundNames} \) of \( \text{BindingPropertyList} \).
2. Append to \( \textit{names} \) the elements of \( \text{BoundNames} \) of \( \text{BindingRestProperty} \).
3. Return \( \textit{names} \).

**ArrayBindingPattern** : \[ \text{Elision}_{\text{opt}} \]

1. Return a new empty \( \text{List} \).

**ArrayBindingPattern** : \[ \text{Elision}_{\text{opt}} \ \text{BindingRestElement} \]

1. Return the \( \text{BoundNames} \) of \( \text{BindingRestElement} \).

**ArrayBindingPattern** : \[ \text{BindingElementList} , \text{Elision}_{\text{opt}} \]

1. Return the \( \text{BoundNames} \) of \( \text{BindingElementList} \).

**ArrayBindingPattern** : \[ \text{BindingElementList} , \text{Elision}_{\text{opt}} \ \text{BindingRestElement} \]

1. Let \( \textit{names} \) be \( \text{BoundNames} \) of \( \text{BindingElementList} \).
2. Append to \( \textit{names} \) the elements of \( \text{BoundNames} \) of \( \text{BindingRestElement} \).
3. Return \( \textit{names} \).

**BindingPropertyList** : \( \text{BindingPropertyList} , \text{BindingProperty} \)

1. Let \( \textit{names} \) be \( \text{BoundNames} \) of \( \text{BindingPropertyList} \).
2. Append to \( \textit{names} \) the elements of \( \text{BoundNames} \) of \( \text{BindingProperty} \).
3. Return \( \textit{names} \).

**BindingElementList** : \( \text{BindingElementList} , \text{BindingElisionElement} \)

1. Let \( \textit{names} \) be \( \text{BoundNames} \) of \( \text{BindingElementList} \).
2. Append to \( \textit{names} \) the elements of \( \text{BoundNames} \) of \( \text{BindingElisionElement} \).
3. Return \( \textit{names} \).

**BindingElisionElement** : \( \text{Elision}_{\text{opt}} \ \text{BindingElement} \)
1. Return **BoundNames** of **BindingElement**.

**BindingProperty**: **PropertyName**: **BindingElement**

1. Return the **BoundNames** of **BindingElement**.

**SingleNameBinding**: **BindingIdentifier** **Initializer**

1. Return the **BoundNames** of **BindingIdentifier**.

**BindingElement**: **BindingPattern** **Initializer**

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 **BoundNames** of **BindingElement**.
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 : { }

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 : { }

CaseBlock : { CaseClausesopt DefaultClause CaseClausesopt }

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 : StatementListopt

1. If the StatementList is present, return the LexicallyDeclaredNames of StatementList.
2. Return a new empty List.

DefaultClause : default : StatementListopt

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 : StatementList

1. Return LexicallyDeclaredNames of StatementList.

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

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

1. If the StatementList is present, return the LexicallyScopedDeclarations of StatementList.
2. Return a new empty List.

**DefaultClause** : default : StatementList

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: \( \text{for} \ ( \text{var} \ \text{VariableDeclarationList} \ \text{opt} \ ; \ \text{Expression}_{\text{opt}} \ ; \ \text{Expression}_{\text{opt}} \ ) \ \text{Statement} \)

1. Let \( names \) be \( \text{BoundNames} \) of \( \text{VariableDeclarationList} \).
2. Append to \( names \) the elements of the \( \text{VarDeclaredNames} \) of \( \text{Statement} \).
3. Return \( names \).

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

1. Return the \( \text{VarDeclaredNames} \) of \( \text{Statement} \).

ForInOfStatement:

- \( \text{for} \ ( \ \text{LeftHandSideExpression} \ \text{in} \ \text{Expression} \ ) \ \text{Statement} \)
- \( \text{for} \ ( \ \text{ForDeclaration} \ \text{in} \ \text{Expression} \ ) \ \text{Statement} \)
- \( \text{for} \ ( \ \text{LeftHandSideExpression} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)
- \( \text{for} \ ( \ \text{ForDeclaration} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)
- \( \text{for await} \ ( \ \text{LeftHandSideExpression} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)
- \( \text{for await} \ ( \ \text{ForDeclaration} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)

1. Return the \( \text{VarDeclaredNames} \) of \( \text{Statement} \).

ForInOfStatement:

- \( \text{for} \ ( \ \text{var} \ \text{ForBinding} \ \text{in} \ \text{Expression} \ ) \ \text{Statement} \)
- \( \text{for} \ ( \ \text{var} \ \text{ForBinding} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)
- \( \text{for await} \ ( \ \text{var} \ \text{ForBinding} \ \text{of} \ \text{AssignmentExpression} \ ) \ \text{Statement} \)

1. Let \( names \) be the \( \text{BoundNames} \) of \( \text{ForBinding} \).
2. Append to \( names \) the elements of the \( \text{VarDeclaredNames} \) of \( \text{Statement} \).
3. Return \( names \).

NOTE This section is extended by Annex B.3.6.

WithStatement: \( \text{with} \ ( \ \text{Expression} \ ) \ \text{Statement} \)

1. Return the \( \text{VarDeclaredNames} \) of \( \text{Statement} \).

SwitchStatement: \( \text{switch} \ ( \ \text{Expression} \ ) \ \text{CaseBlock} \)

1. Return the \( \text{VarDeclaredNames} \) of \( \text{CaseBlock} \).

CaseBlock: \{ \}

1. Return a new empty \( \text{List} \).

CaseBlock: \{ \ \text{CaseClauses}_{\text{opt}} \ \text{DefaultClause} \ \text{CaseClauses}_{\text{opt}} \}

1. If the first \( \text{CaseClauses} \) is present, let \( names \) be the \( \text{VarDeclaredNames} \) of the first \( \text{CaseClauses} \).
2. Else, let \( names \) be a new empty \( \text{List} \).
3. Append to \( names \) the elements of the \( \text{VarDeclaredNames} \) of \( \text{DefaultClause} \).
4. If the second \( \text{CaseClauses} \) is not present, return \( names \).
5. Return the result of appending to \( names \) the elements of the \( \text{VarDeclaredNames} \) of the second \( \text{CaseClauses} \).
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`:

   ```
   LabelledItem : LabelledItem
   ```
1. Return the \texttt{VarScopedDeclarations} of \textit{LabelledItem}.

\textit{LabelledItem} : \texttt{FunctionDeclaration}

1. Return a new empty \texttt{List}.

\textit{TryStatement} : \texttt{try} Block \texttt{Catch}

1. Let \textit{declarations} be \texttt{VarScopedDeclarations} of \textit{Block}.
2. Append to \textit{declarations} the elements of the \texttt{VarScopedDeclarations} of \textit{Catch}.
3. Return \textit{declarations}.

\textit{TryStatement} : \texttt{try} Block \texttt{Finally}

1. Let \textit{declarations} be \texttt{VarScopedDeclarations} of \textit{Block}.
2. Append to \textit{declarations} the elements of the \texttt{VarScopedDeclarations} of \textit{Finally}.
3. Return \textit{declarations}.

\textit{TryStatement} : \texttt{try} Block \texttt{Catch} \texttt{Finally}

1. Let \textit{declarations} be \texttt{VarScopedDeclarations} of \textit{Block}.
2. Append to \textit{declarations} the elements of the \texttt{VarScopedDeclarations} of \textit{Catch}.
3. Append to \textit{declarations} the elements of the \texttt{VarScopedDeclarations} of \textit{Finally}.
4. Return \textit{declarations}.

\textit{Catch} : \texttt{catch} ( \texttt{CatchParameter} ) Block

1. Return the \texttt{VarScopedDeclarations} of \textit{Block}.

\textit{FunctionStatementList} : [\texttt{empty}]

1. Return a new empty \texttt{List}.

\textit{FunctionStatementList} : \texttt{StatementList}

1. Return the \texttt{TopLevelVarScopedDeclarations} of \texttt{StatementList}.

\textit{ConciseBody} : \texttt{ExpressionBody}

1. Return a new empty \texttt{List}.

\textit{AsyncConciseBody} : \texttt{ExpressionBody}

1. Return a new empty \texttt{List}.

\textit{ScriptBody} : \texttt{StatementList}

1. Return \texttt{TopLevelVarScopedDeclarations} of \texttt{StatementList}.

\textit{Module} : [\texttt{empty}]

1. Return a new empty \texttt{List}.

\textit{ModuleItemList} : \texttt{ModuleItemList \ ModuleItem}

1. Let \textit{declarations} be \texttt{VarScopedDeclarations} of \texttt{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

StatementListItem : Statement

StatementListItem : Declaration

LabelledStatement : LabelIdentifier : LabelledItem

LabelledStatement : LabelIdentifier

StatementListItem : HoistableDeclaration

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

8.1.9 Static Semantics: TopLevelLexicallyScopedDeclarations

Block : { }

Block : {

StatementList : StatementList StatementListItem

StatementListItem : Statement

StatementListItem : VariableDeclaration

StatementListItem : HoistableDeclaration
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 hasDuplicates be ContainsDuplicateLabels of StatementList with argument labelSet.
2. If hasDuplicates 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 hasDuplicates be ContainsDuplicateLabels of the first CaseClauses with argument labelSet.
   b. If hasDuplicates is true, return true.
2. Let hasDuplicates be ContainsDuplicateLabels of DefaultClause with argument labelSet.
3. If hasDuplicates 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 hasDuplicates be ContainsDuplicateLabels of CaseClauses with argument labelSet.
2. If hasDuplicates 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 hasDuplicates be ContainsDuplicateLabels of Block with argument labelSet.
2. If hasDuplicates is true, return true.
3. Return ContainsDuplicateLabels of Catch with argument labelSet.

TryStatement : try  Block  Finally

1. Let hasDuplicates be ContainsDuplicateLabels of Block with argument labelSet.
2. If hasDuplicates 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** :

```plaintext
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** :

```plaintext
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 : \texttt{with ( Expression ) Statement}

1. Return \texttt{ContainsUndefinedBreakTarget} of \texttt{Statement} with argument \texttt{labelSet}.

SwitchStatement : \texttt{switch ( Expression ) CaseBlock}

1. Return \texttt{ContainsUndefinedBreakTarget} of \texttt{CaseBlock} with argument \texttt{labelSet}.

CaseBlock : \{ \}

1. Return \texttt{false}.

CaseBlock : \{ CaseClauses\_{opt} DefaultClause CaseClauses\_{opt} \}

1. If the first \texttt{CaseClauses} is present, then
   a. Let \texttt{hasUndefinedLabels} be \texttt{ContainsUndefinedBreakTarget} of the first \texttt{CaseClauses} with argument \texttt{labelSet}.
   b. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
2. Let \texttt{hasUndefinedLabels} be \texttt{ContainsUndefinedBreakTarget} of \texttt{DefaultClause} with argument \texttt{labelSet}.
3. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
4. If the second \texttt{CaseClauses} is not present, return \texttt{false}.
5. Return \texttt{ContainsUndefinedBreakTarget} of the second \texttt{CaseClauses} with argument \texttt{labelSet}.

CaseClauses : CaseClauses CaseClause

1. Let \texttt{hasUndefinedLabels} be \texttt{ContainsUndefinedBreakTarget} of \texttt{CaseClauses} with argument \texttt{labelSet}.
2. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
3. Return \texttt{ContainsUndefinedBreakTarget} of \texttt{CaseClause} with argument \texttt{labelSet}.

CaseClause : case Expression : StatementList\_{opt}

1. If the \texttt{StatementList} is present, return \texttt{ContainsUndefinedBreakTarget} of \texttt{StatementList} with argument \texttt{labelSet}.
2. Return \texttt{false}.

DefaultClause : default : StatementList\_{opt}

1. If the \texttt{StatementList} is present, return \texttt{ContainsUndefinedBreakTarget} of \texttt{StatementList} with argument \texttt{labelSet}.
2. Return \texttt{false}.

LabelledStatement : LabelIdentifier : LabelledItem

1. Let \texttt{label} be the \texttt{StringValue} of \texttt{LabelIdentifier}.
2. Let \texttt{newLabelSet} be a copy of \texttt{labelSet} with \texttt{label} appended.
3. Return \texttt{ContainsUndefinedBreakTarget} of \texttt{LabelledItem} with argument \texttt{newLabelSet}.

LabelledItem : FunctionDeclaration

1. Return \texttt{false}.

TryStatement : \texttt{try Block Catch}

1. Let \texttt{hasUndefinedLabels} be \texttt{ContainsUndefinedBreakTarget} of \texttt{Block} with argument \texttt{labelSet}.
2. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
3. Return \texttt{ContainsUndefinedBreakTarget} of \texttt{Catch} with argument \texttt{labelSet}.
TryStatement: \texttt{try} Block Finally

1. Let $\text{hasUndefinedLabels}$ be $\text{ContainsUndefinedBreakTarget}$ of Block with argument $\text{labelSet}$.
2. If $\text{hasUndefinedLabels}$ is \texttt{true}, return \texttt{true}.
3. Return $\text{ContainsUndefinedBreakTarget}$ of Finally with argument $\text{labelSet}$.

TryStatement: \texttt{try} Block Catch Finally

1. Let $\text{hasUndefinedLabels}$ be $\text{ContainsUndefinedBreakTarget}$ of Block with argument $\text{labelSet}$.
2. If $\text{hasUndefinedLabels}$ is \texttt{true}, return \texttt{true}.
3. Let $\text{hasUndefinedLabels}$ be $\text{ContainsUndefinedBreakTarget}$ of Catch with argument $\text{labelSet}$.
4. If $\text{hasUndefinedLabels}$ is \texttt{true}, return \texttt{true}.
5. Return $\text{ContainsUndefinedBreakTarget}$ of Finally with argument $\text{labelSet}$.

Catch: \texttt{catch (CatchParameter)} Block

1. Return $\text{ContainsUndefinedBreakTarget}$ of Block with argument $\text{labelSet}$.

FunctionStatementList: [empty]

1. Return \texttt{false}.

ModuleItemList: ModuleItemList ModuleItem

1. Let $\text{hasUndefinedLabels}$ be $\text{ContainsUndefinedBreakTarget}$ of ModuleItemList with argument $\text{labelSet}$.
2. If $\text{hasUndefinedLabels}$ is \texttt{true}, return \texttt{true}.
3. Return $\text{ContainsUndefinedBreakTarget}$ of ModuleItem with argument $\text{labelSet}$.

ModuleItem:

- \texttt{ImportDeclaration}
- \texttt{ExportDeclaration}

1. Return \texttt{false}.

8.2.3 Static Semantics: ContainsUndefinedContinueTarget

With parameters $\text{iterationSet}$ and $\text{labelSet}$.

Statement:

- \texttt{VariableStatement}
- \texttt{EmptyStatement}
- \texttt{ExpressionStatement}
- \texttt{BreakStatement}
- \texttt{ReturnStatement}
- \texttt{ThrowStatement}
- \texttt{DebuggerStatement}

Block: \{ \}

StatementListItem: Declaration

1. Return \texttt{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 « ».
NOTE: This section is extended by Annex B.3.6.

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 CoverParenthesizedExpression of CoverParenthesizedExpressionAndArrowParameterList.
2. If IsFunctionDefinition of \( expr \) is false, return false.
3. Return HasName of \( expr \).

FunctionExpression :

\[
\text{function}\ (\ FormalParameters\ )\ \{\ \text{FunctionBody}\ \}
\]

GeneratorExpression :

\[
\text{function}\ *\ (\ FormalParameters\ )\ \{\ \text{GeneratorBody}\ \}
\]

AsyncGeneratorExpression :

\[
\text{async}\ \text{function}\ *\ (\ FormalParameters\ )\ \{\ \text{AsyncGeneratorBody}\ \}
\]

AsyncFunctionExpression :

\[
\text{async}\ \text{function}\ (\ FormalParameters\ )\ \{\ \text{AsyncFunctionBody}\ \}
\]

ArrowFunction :

\[
\text{ArrowParameters} \Rightarrow \text{ConciseBody}
\]

AsyncArrowFunction :

\[
\text{async}\ \text{AsyncArrowBindingIdentifier} \Rightarrow \text{AsyncConciseBody}
\]

CoverCallExpressionAndAsyncArrowHead \Rightarrow \text{AsyncConciseBody}

ClassExpression : \text{class} \ ClassTail

1. Return false.

FunctionExpression :

\[
\text{function}\ BindingIdentifier\ (\ FormalParameters\ )\ \{\ \text{FunctionBody}\ \}
\]

GeneratorExpression :

\[
\text{function}\ *\ BindingIdentifier\ (\ FormalParameters\ )\ \{\ \text{GeneratorBody}\ \}
\]

AsyncGeneratorExpression :

\[
\text{async}\ \text{function}\ *\ BindingIdentifier\ (\ FormalParameters\ )\ \{\ \text{AsyncGeneratorBody}\ \}
\]

AsyncFunctionExpression :

\[
\text{async}\ \text{function}\ BindingIdentifier\ (\ FormalParameters\ )\ \{\ \text{AsyncFunctionBody}\ \}
\]

ClassExpression : \text{class} \ BindingIdentifier\ ClassTail

1. Return true.

8.3.2 Static Semantics: IsFunctionDefinition
PrimaryExpression : CoverParenthesizedExpressionAndArrowParameterList

1. Let \( expr \) be CoverParenthesizedExpression of CoverParenthesizedExpressionAndArrowParameterList.
2. Return IsFunctionDefinition of \( expr \).

PrimaryExpression :

\[
\text{this}
\]

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 \texttt{true}.

### 8.3.3 Static Semantics: \texttt{IsAnonymousFunctionDefinition ( expr )}

The abstract operation \texttt{IsAnonymousFunctionDefinition} takes argument \texttt{expr} (a Parse Node for \texttt{AssignmentExpression} or a Parse Node for \texttt{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 \texttt{IsFunctionDefinition of expr} is \texttt{false}, return \texttt{false}.
2. Let \texttt{hasName} be \texttt{HasName of expr}.
3. If \texttt{hasName} is \texttt{true}, return \texttt{false}.
4. Return \texttt{true}.

### 8.3.4 Static Semantics: \texttt{IsIdentifierRef}

\texttt{PrimaryExpression} : \texttt{IdentifierReference}

1. Return \texttt{true}.

\texttt{PrimaryExpression} :
\begin{itemize}
\item \texttt{this}
\item \texttt{Literal}
\item \texttt{ArrayLiteral}
\item \texttt{ObjectLiteral}
\item \texttt{FunctionExpression}
\item \texttt{ClassExpression}
\item \texttt{GeneratorExpression}
\item \texttt{AsyncFunctionExpression}
\item \texttt{AsyncGeneratorExpression}
\item \texttt{RegularExpressionLiteral}
\item \texttt{TemplateLiteral}
\item \texttt{CoverParenthesizedExpressionAndArrowParameterList}
\end{itemize}

\texttt{MemberExpression} :
\begin{itemize}
\item \texttt{MemberExpression [ Expression ]}
\item \texttt{MemberExpression . IdentifierName}
\item \texttt{MemberExpression TemplateLiteral}
\item \texttt{SuperProperty}
\item \texttt{MetaProperty}
\item \texttt{new MemberExpression Arguments}
\end{itemize}

\texttt{NewExpression} :
\begin{itemize}
\item \texttt{new NewExpression}
\end{itemize}

\texttt{LeftHandSideExpression} :
\begin{itemize}
\item \texttt{CallExpression}
\item \texttt{OptionalExpression}
\end{itemize}

1. Return \texttt{false}.
8.3.5 Runtime Semantics: NamedEvaluation

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

**PrimaryExpression**: \( \text{CoverParenthesizedExpressionAndArrowParameterList} \)

1. Let \( \text{expr} \) be \( \text{CoveredParenthesizedExpression} \) of \( \text{CoverParenthesizedExpressionAndArrowParameterList} \).  
2. Return the result of performing \( \text{NamedEvaluation} \) for \( \text{expr} \) with argument \( \text{name} \).

**ParenthesizedExpression**: \((\text{Expression})\)

1. Assert: \( \text{IsAnonymousFunctionDefinition(Expression)} \) is \( \text{true} \).  
2. Return the result of performing \( \text{NamedEvaluation} \) for \( \text{Expression} \) with argument \( \text{name} \).

**FunctionExpression**: \( \text{function } (\text{FormalParameters}) \{ \text{FunctionBody} \} \)

1. Return \( \text{InstantiateOrdinaryFunctionExpression} \) of \( \text{FunctionExpression} \) with argument \( \text{name} \).

**GeneratorExpression**: \( \text{function } * (\text{FormalParameters}) \{ \text{GeneratorBody} \} \)

1. Return \( \text{InstantiateGeneratorFunctionExpression} \) of \( \text{GeneratorExpression} \) with argument \( \text{name} \).

**AsyncGeneratorExpression**: \( \text{async function } * (\text{FormalParameters}) \{ \text{AsyncGeneratorBody} \} \)

1. Return \( \text{InstantiateAsyncGeneratorFunctionExpression} \) of \( \text{AsyncGeneratorExpression} \) with argument \( \text{name} \).

**AsyncFunctionExpression**: \( \text{async function } (\text{FormalParameters}) \{ \text{AsyncFunctionBody} \} \)

1. Return \( \text{InstantiateAsyncFunctionExpression} \) of \( \text{AsyncFunctionExpression} \) with argument \( \text{name} \).

**ArrowFunction**: \( \text{ArrowParameters} \Rightarrow \text{ConciseBody} \)

1. Return \( \text{InstantiateArrowFunctionExpression} \) of \( \text{ArrowFunction} \) with argument \( \text{name} \).

**AsyncArrowFunction**:

\[
\begin{align*}
\text{async } \text{AsyncArrowBindingIdentifier} & \Rightarrow \text{AsyncConciseBody} \\
\text{CoverCallExpressionAndAsyncArrowHead} & \Rightarrow \text{AsyncConciseBody}
\end{align*}
\]

1. Return \( \text{InstantiateAsyncArrowFunctionExpression} \) of \( \text{AsyncArrowFunction} \) with argument \( \text{name} \).

**ClassExpression**: \( \text{class } \text{ClassTail} \)

1. Let \( \text{value} \) be the result of \( \text{ClassDefinitionEvaluation} \) of \( \text{ClassTail} \) with arguments \( \text{undefined} \) and \( \text{name} \).  
2. \( \text{ReturnIfAbrupt(value)} \).  
3. Set \( \text{value}[[\text{SourceText}]] \) to the source text matched by \( \text{ClassExpression} \).  
4. Return \( \text{value} \).

8.4 Contains

8.4.1 Static Semantics: Contains

With parameter \( \text{symbol} \).
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 \( \text{child} \) of this Parse Node, do
   a. If \( \text{child} \) is an instance of \( \text{symbol} \), return \( \text{true} \).
   b. If \( \text{child} \) is an instance of a nonterminal, then
      i. Let \( \text{contained} \) be the result of \( \text{child} \) Contains \( \text{symbol} \).
      ii. If \( \text{contained} \) is \( \text{true} \), return \( \text{true} \).
2. Return \( \text{false} \).

**FunctionDeclaration**:

\[
\begin{align*}
\text{function} & \quad \text{BindingIdentifier} \ ( \text{FormalParameters} 
\{ \text{FunctionBody} \} \\
\text{function} & \quad ( \text{FormalParameters} 
\{ \text{FunctionBody} \}
\end{align*}
\]

**FunctionExpression**:

\[
\begin{align*}
\text{function} & \quad \text{BindingIdentifier}_{\text{opt}} \ ( \text{FormalParameters} 
\{ \text{FunctionBody} \} \\
\text{function} & \quad ( \text{FormalParameters} 
\{ \text{FunctionBody} \}
\end{align*}
\]

**GeneratorDeclaration**:

\[
\begin{align*}
\text{function} & \quad * \ \text{BindingIdentifier} \ ( \text{FormalParameters} 
\{ \text{GeneratorBody} \} \\
\text{function} & \quad * \ ( \text{FormalParameters} 
\{ \text{GeneratorBody} \}
\end{align*}
\]

**GeneratorExpression**:

\[
\begin{align*}
\text{function} & \quad * \ \text{BindingIdentifier}_{\text{opt}} \ ( \text{FormalParameters} 
\{ \text{GeneratorBody} \} \\
\text{function} & \quad * \ ( \text{FormalParameters} 
\{ \text{GeneratorBody} \}
\end{align*}
\]

**AsyncGeneratorDeclaration**:

\[
\begin{align*}
\text{async function} & \quad * \ \text{BindingIdentifier} \ ( \text{FormalParameters} 
\{ \text{AsyncGeneratorBody} \} \\
\text{async function} & \quad * \ ( \text{FormalParameters} 
\{ \text{AsyncGeneratorBody} \}
\end{align*}
\]

**AsyncGeneratorExpression**:

\[
\begin{align*}
\text{async function} & \quad * \ \text{BindingIdentifier}_{\text{opt}} \ ( \text{FormalParameters} 
\{ \text{AsyncGeneratorBody} \} \\
\text{async function} & \quad * \ ( \text{FormalParameters} 
\{ \text{AsyncGeneratorBody} \}
\end{align*}
\]

1. Return \( \text{false} \).

**NOTE 1** Static semantic rules that depend upon substructure generally do not look into function definitions.

**ClassTail**:

\[
\text{ClassTail} : \quad \text{ClassHeritage}_{\text{opt}} \ {\text{ClassBody}}
\]

1. If \( \text{symbol} \) is \( \text{ClassBody} \), return \( \text{true} \).
2. If \( \text{symbol} \) is \( \text{ClassHeritage} \), then
   a. If \( \text{ClassHeritage} \) is present, return \( \text{true} \); otherwise return \( \text{false} \).
3. Let \( \text{inHeritage} \) be \( \text{ClassHeritage} \) Contains \( \text{symbol} \).
4. If \( \text{inHeritage} \) is \( \text{true} \), return \( \text{true} \).
5. Return the result of ComputedPropertyContains for \( \text{ClassBody} \) with argument \( \text{symbol} \).
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 :

```javascript
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 \texttt{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 \( \text{RequireObjectCoercible}(\text{value}) \).
2. Return the result of performing \( \text{BindingInitialization} \) for \( \text{ObjectBindingPattern} \) using \( \text{value} \) and \( \text{environment} \) as arguments.

**BindingPattern**: \( \text{ArrayBindingPattern} \)

1. Let \( \text{iteratorRecord} \) be \( \text{GetIterator}(\text{value}) \).
2. Let \( \text{result} \) be \( \text{IteratorBindingInitialization} \) of \( \text{ArrayBindingPattern} \) with arguments \( \text{iteratorRecord} \) and \( \text{environment} \).
3. If \( \text{iteratorRecord}.[[\text{Done}]] \) is \( \text{false} \), return \( \text{IteratorClose}(\text{iteratorRecord}, \text{result}) \).
4. Return \( \text{result} \).

**ObjectBindingPattern**: \{ \}

1. Return \( \text{NormalCompletion}(\text{empty}) \).

**ObjectBindingPattern**: \{ \text{BindingPropertyList} \}

1. Perform \( \text{PropertyBindingInitialization} \) for \( \text{BindingPropertyList} \) using \( \text{value} \) and \( \text{environment} \) as the arguments.
2. Return \( \text{NormalCompletion}(\text{empty}) \).

**ObjectBindingPattern**: \{ \text{BindingRestProperty} \}

1. Let \( \text{excludedNames} \) be a new empty \text{List}.
2. Return the result of performing \( \text{RestBindingInitialization} \) of \( \text{BindingRestProperty} \) with \( \text{value} \), \( \text{environment} \), and \( \text{excludedNames} \) as the arguments.

**ObjectBindingPattern**: \{ \text{BindingPropertyList} , \text{BindingRestProperty} \}

1. Let \( \text{excludedNames} \) be \( \text{PropertyBindingInitialization} \) of \( \text{BindingPropertyList} \) with arguments \( \text{value} \) and \( \text{environment} \).
2. Return the result of performing \( \text{RestBindingInitialization} \) of \( \text{BindingRestProperty} \) with arguments \( \text{value} \), \( \text{environment} \), and \( \text{excludedNames} \).

### 8.5.2.1 InitializeBoundName( \( \text{name} \), \( \text{value} \), \( \text{environment} \) )

The abstract operation \( \text{InitializeBoundName} \) takes arguments \( \text{name} \), \( \text{value} \), and \( \text{environment} \). It performs the following steps when called:

1. \( \text{Assert: Type}(\text{name}) \) is \text{String}.
2. If \( \text{environment} \) is not \text{undefined}, then
   a. Perform \( \text{environment}.\text{InitializeBinding}(\text{name}, \text{value}) \).
   b. Return \( \text{NormalCompletion}(\text{undefined}) \).
3. Else,
   a. Let \( \text{lhs} \) be \( \text{ResolveBinding}(\text{name}) \).
   b. Return \( \text{? PutValue}(\text{lhs}, \text{value}) \).

### 8.5.3 Runtime Semantics: IteratorBindingInitialization

With parameters \( \text{iteratorRecord} \) and \( \text{environment} \).
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.

SetNameBinding : 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 $\text{IteratorValue}(\text{next})$.
   b. If $v$ is an abrupt completion, set $\text{iteratorRecord}[[\text{Done}]]$ to $\text{true}$.
   c. $\text{ReturnIfAbrupt}(v)$.
7. If $\text{iteratorRecord}[[\text{Done}]]$ is $\text{true}$, let $v$ be $\text{undefined}$.
8. Return the result of performing $\text{BindingInitialization}$ for $\text{BindingIdentifier}$ using $v$ and $\text{environment}$ as the arguments.

**ArrowParameters**: $\text{CoverParenthesizedExpressionAndArrowParameterList}$

1. Let $\text{formals}$ be $\text{CoveredFormalsList}$ of $\text{CoverParenthesizedExpressionAndArrowParameterList}$.
2. Return $\text{IteratorBindingInitialization}$ of $\text{formals}$ with arguments $\text{iteratorRecord}$ and $\text{environment}$.

**AsyncArrowBindingIdentifier**: $\text{BindingIdentifier}$

1. Assert: $\text{iteratorRecord}[[\text{Done}]]$ is $\text{false}$.
2. Let $\text{next}$ be $\text{IteratorStep}(\text{iteratorRecord})$.
3. If $\text{next}$ is an abrupt completion, set $\text{iteratorRecord}[[\text{Done}]]$ to $\text{true}$.
4. $\text{ReturnIfAbrupt}(\text{next})$.
5. If $\text{next}$ is $\text{false}$, set $\text{iteratorRecord}[[\text{Done}]]$ to $\text{true}$.
6. Else,
   a. Let $v$ be $\text{IteratorValue}(\text{next})$.
   b. If $v$ is an abrupt completion, set $\text{iteratorRecord}[[\text{Done}]]$ to $\text{true}$.
   c. $\text{ReturnIfAbrupt}(v)$.
7. If $\text{iteratorRecord}[[\text{Done}]]$ is $\text{true}$, let $v$ be $\text{undefined}$.
8. Return the result of performing $\text{BindingInitialization}$ for $\text{BindingIdentifier}$ using $v$ and $\text{environment}$ as the arguments.

**8.5.4 Static Semantics: AssignmentTargetType**

**IdentifierReference**: $\text{Identifier}$

1. If this $\text{IdentifierReference}$ is contained in strict mode code and $\text{StringValue}$ of $\text{Identifier}$ is "$\text{eval}$" or "$\text{arguments}$", return invalid.
2. Return simple.

**IdentifierReference**: $\text{yield}$

**IdentifierReference**: $\text{await}$

**CallExpression**: $\text{CallExpression}$

1. Return simple.

**MemberExpression**: $\text{MemberExpression}$

1. Return simple.
CoverParenthesizedExpressionAndArrowParameterList

1. Let \( expr \) be \( \text{CoveredParenthesizedExpression} \) of \( \text{CoverParenthesizedExpressionAndArrowParameterList} \).
2. Return \( \text{AssignmentTargetType} \) of \( expr \).

**PrimaryExpression:**

\[ \text{this} \]
\[ \text{Literal} \]
\[ \text{ArrayLiteral} \]
\[ \text{ObjectLiteral} \]
\[ \text{FunctionExpression} \]
\[ \text{ClassExpression} \]
\[ \text{GeneratorExpression} \]
\[ \text{AsyncFunctionExpression} \]
\[ \text{AsyncGeneratorExpression} \]
\[ \text{RegularExpressionLiteral} \]
\[ \text{TemplateLiteral} \]

**CallExpression:**

\[ \text{CoverCallExpressionAndAsyncArrowHead} \]
\[ \text{SuperCall} \]
\[ \text{ImportCall} \]
\[ \text{CallExpression} \text{ Arguments} \]
\[ \text{CallExpression} \text{ TemplateLiteral} \]

**NewExpression:**

\[ \text{new} \text{ NewExpression} \]

**MemberExpression:**

\[ \text{MemberExpression} \text{ TemplateLiteral} \]
\[ \text{new} \text{ MemberExpression} \text{ Arguments} \]

**NewTarget:**

\[ \text{new} . \text{target} \]

**ImportMeta:**

\[ \text{import} . \text{meta} \]

**LeftHandSideExpression:**

\[ \text{OptionalExpression} \]

**UpdateExpression:**

\[ \text{LeftHandSideExpression} \text{ ++} \]
\[ \text{LeftHandSideExpression} \text{ --} \]
\[ ++ \text{ UnaryExpression} \]
\[ -- \text{ UnaryExpression} \]

**UnaryExpression:**

\[ \text{delete} \text{ UnaryExpression} \]
\[ \text{void} \text{ UnaryExpression} \]
\[ \text{typeof} \text{ UnaryExpression} \]
\[ + \text{ UnaryExpression} \]
\[ - \text{ UnaryExpression} \]
~ 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
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.
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>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasBinding(N)</td>
<td>Determine if an Environment Record has a binding for the String value N. Return <strong>true</strong> if it does and <strong>false</strong> if it does not.</td>
</tr>
<tr>
<td>CreateMutableBinding(N, D)</td>
<td>Create a new but uninitialized mutable binding in an Environment Record. The String value N is the text of the bound name. If the Boolean argument D is <strong>true</strong> the binding may be subsequently deleted.</td>
</tr>
<tr>
<td>CreateImmutableBinding(N, S)</td>
<td>Create a new but uninitialized immutable binding in an Environment Record. The String value N is the text of the bound name. If S is <strong>true</strong> 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.</td>
</tr>
<tr>
<td>InitializeBinding(N, V)</td>
<td>Set the value of an already existing but uninitialized binding in an Environment Record. The String value N is the text of the bound name. V is the value for the binding and is a value of any ECMAScript language type.</td>
</tr>
<tr>
<td>SetMutableBinding(N, V, S)</td>
<td>Set the value of an already existing mutable binding in an Environment Record. The String value N is the text of the bound name. V is the value for the binding and may be a value of any ECMAScript language type. S is a Boolean flag. If S is <strong>true</strong> and the binding cannot be set throw a <strong>TypeError</strong> exception.</td>
</tr>
<tr>
<td>GetBindingValue(N, S)</td>
<td>Returns the value of an already existing binding from an Environment Record. The String value N is the text of the bound name. S is used to identify references originating in strict mode code or that otherwise require strict mode reference semantics. If S is <strong>true</strong> and the binding does not exist throw a <strong>ReferenceError</strong> exception. If the binding exists but is uninitialized a <strong>ReferenceError</strong> is thrown, regardless of the value of S.</td>
</tr>
<tr>
<td>DeleteBinding(N)</td>
<td>Delete a binding from an Environment Record. The String value N is the text of the bound name. If a binding for N exists, remove the binding and return <strong>true</strong>. If the binding exists but cannot be removed return <strong>false</strong>. If the binding does not exist return <strong>true</strong>.</td>
</tr>
<tr>
<td>HasThisBinding()</td>
<td>Determine if an Environment Record establishes a this binding. Return <strong>true</strong> if it does and <strong>false</strong> if it does not.</td>
</tr>
<tr>
<td>HasSuperBinding()</td>
<td>Determine if an Environment Record establishes a super method binding. Return <strong>true</strong> if it does and <strong>false</strong> if it does not.</td>
</tr>
<tr>
<td>WithBaseObject()</td>
<td>If this Environment Record is associated with a with statement, return the with object. Otherwise, return <strong>undefined</strong>.</td>
</tr>
</tbody>
</table>

### 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 HasBinding (N)

The HasBinding concrete method of a declarative Environment Record \texttt{envRec} takes argument \textit{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 \texttt{envRec} has a binding for the name that is the value of \textit{N}, return \texttt{true}.
2. Return \texttt{false}.

9.1.1.2 CreateMutableBinding (N, D)

The CreateMutableBinding concrete method of a declarative Environment Record \texttt{envRec} takes arguments \textit{N} (a String) and \textit{D} (a Boolean). It creates a new mutable binding for the name \textit{N} that is uninitialized. A binding must not already exist in this Environment Record for \textit{N}. If \textit{D} has the value \texttt{true}, the new binding is marked as being subject to deletion. It performs the following steps when called:

1. Assert: \texttt{envRec} does not already have a binding for \textit{N}.
2. Create a mutable binding in \texttt{envRec} for \textit{N} and record that it is uninitialized. If \textit{D} is \texttt{true}, record that the newly created binding may be deleted by a subsequent DeleteBinding call.
3. Return NormalCompletion(\texttt{empty}).

9.1.1.3 CreateImmutableBinding (N, S)

The CreateImmutableBinding concrete method of a declarative Environment Record \texttt{envRec} takes arguments \textit{N} (a String) and \textit{S} (a Boolean). It creates a new immutable binding for the name \textit{N} that is uninitialized. A binding must not already exist in this Environment Record for \textit{N}. If \textit{S} has the value \texttt{true}, the new binding is marked as a strict binding. It performs the following steps when called:

1. Assert: \texttt{envRec} does not already have a binding for \textit{N}.
2. Create an immutable binding in \texttt{envRec} for \textit{N} and record that it is uninitialized. If \textit{S} is \texttt{true}, record that the newly created binding is a strict binding.
3. Return NormalCompletion(\texttt{empty}).

9.1.1.4 InitializeBinding (N, V)

The InitializeBinding concrete method of a declarative Environment Record \texttt{envRec} takes arguments \textit{N} (a String) and \textit{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 \textit{N} to the value of argument \textit{V}. An uninitialized binding for \textit{N} must already exist. It performs the following steps when called:

1. Assert: \texttt{envRec} must have an uninitialized binding for \textit{N}.
2. Set the bound value for \textit{N} in \texttt{envRec} to \textit{V}.
3. Record that the binding for \textit{N} in \texttt{envRec} has been initialized.
4. Return NormalCompletion(\texttt{empty}).

9.1.1.5 SetMutableBinding (N, V, S)

The SetMutableBinding concrete method of a declarative Environment Record \texttt{envRec} takes arguments \textit{N} (a String), \textit{V} (an ECMAScript language value) and \textit{S} (a Boolean). It is used to set the bound value of a mutable binding. It performs the following steps when called:

1. Assert: \texttt{envRec} must have a mutable binding for \textit{N}.
2. Set the bound value for \textit{N} in \texttt{envRec} to \textit{V}.
3. Record that the binding for \textit{N} in \texttt{envRec} has been initialized.
4. Return NormalCompletion(\texttt{empty}).
(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 \texttt{TypeError} is thrown if \( S \) is \texttt{true}. It performs the following steps when called:

1. If \( envRec \) does not have a binding for \( N \), then
   a. If \( S \) is \texttt{true}, throw a \texttt{ReferenceError} exception.
   b. Perform \( envRec.CreateCommandBinding(N, \texttt{true}) \).
   c. Perform \( envRec.InitializeBinding(N, V) \).
   d. Return \texttt{NormalCompletion(empty)}.
2. If the binding for \( N \) in \( envRec \) is a strict binding, set \( S \) to \texttt{true}.
3. If the binding for \( N \) in \( envRec \) has not yet been initialized, throw a \texttt{ReferenceError} exception.
4. Else if the binding for \( N \) in \( envRec \) is a mutable binding, change its bound value to \( V \).
5. Else,
   a. \texttt{Assert}: This is an attempt to change the value of an immutable binding.
   b. If \( S \) is \texttt{true}, throw a \texttt{TypeError} exception.
6. Return \texttt{NormalCompletion(empty)}.

NOTE

An example of ECMAScript code that results in a missing binding at step 1 is:

```javascript
function f() { eval("var x; x = (delete x, 0);"); }
```

9.1.1.1.6 GetBindingValue \((N, S)\)

The \texttt{GetBindingValue} concrete method of a \texttt{declarative Environment Record} \texttt{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 \texttt{ReferenceError} is thrown, regardless of the value of \( S \). It performs the following steps when called:

1. \texttt{Assert}: \texttt{envRec} has a binding for \( N \).
2. If the binding for \( N \) in \texttt{envRec} is an uninitialized binding, throw a \texttt{ReferenceError} exception.
3. Return the value currently bound to \( N \) in \texttt{envRec}.

9.1.1.1.7 DeleteBinding \((N)\)

The \texttt{DeleteBinding} concrete method of a \texttt{declarative Environment Record} \texttt{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. \texttt{Assert}: \texttt{envRec} has a binding for the name that is the value of \( N \).
2. If the binding for \( N \) in \texttt{envRec} cannot be deleted, return \texttt{false}.
3. Remove the binding for \( N \) from \texttt{envRec}.
4. Return \texttt{true}.

9.1.1.1.8 HasThisBinding \((\)\)

The \texttt{HasThisBinding} concrete method of a \texttt{declarative Environment Record} \texttt{envRec} takes no arguments. It performs the following steps when called:
1. Return \text{false}.

\textbf{NOTE} \hspace{2cm} A regular \textit{declarative Environment Record} (i.e., one that is neither a \textit{function Environment Record} nor a \textit{module Environment Record}) does not provide a \texttt{this} binding.

\textbf{9.1.1.9 HasSuperBinding ( )}

The \textbf{HasSuperBinding} concrete method of a \textit{declarative Environment Record} \texttt{envRec} takes no arguments. It performs the following steps when called:

1. Return \texttt{false}.

\textbf{NOTE} \hspace{2cm} A regular \textit{declarative Environment Record} (i.e., one that is neither a \textit{function Environment Record} nor a \textit{module Environment Record}) does not provide a \texttt{super} binding.

\textbf{9.1.1.10 WithBaseObject ( )}

The \textbf{WithBaseObject} concrete method of a \textit{declarative Environment Record} \texttt{envRec} takes no arguments. It performs the following steps when called:

1. Return \texttt{undefined}.

\textbf{9.1.1.2 Object Environment Records}

Each \textit{object Environment Record} is associated with an object called its \textit{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 \texttt{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 \texttt{[[ 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 \texttt{false}. Immutable bindings do not exist for object Environment Records.

Object Environment Records created for \texttt{with} statements (14.11) can provide their binding object as an implicit \texttt{this} value for use in function calls. The capability is controlled by a \texttt{withEnvironment} Boolean value that is associated with each object Environment Record. By default, the value of \texttt{withEnvironment} is \texttt{false} for any object Environment Record.

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

\textbf{9.1.1.2.1 HasBinding ( \textit{N} )}

The \textbf{HasBinding} concrete method of an \textit{object Environment Record} \texttt{envRec} takes argument \textit{N} (a String). It determines if its associated binding object has a property whose name is the value of the argument \textit{N}. It performs the following steps when called:

1. Let \texttt{bindings} be the binding object for \texttt{envRec}.
2. Let \texttt{foundBinding} be \texttt{? HasProperty(bindings, N)}.
3. If \texttt{foundBinding} is \texttt{false}, return \texttt{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:


**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.

### 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.

### 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`.

---

**NOTE**

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

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[ThisValue]]</td>
<td>Any</td>
<td>This is the this value used for this invocation of the function.</td>
</tr>
<tr>
<td>[[ThisBindingStatus]]</td>
<td>lexical</td>
<td>If the value is lexical, this is an ArrowFunction and does not have a local this value.</td>
</tr>
<tr>
<td></td>
<td>initialized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uninitialized</td>
<td></td>
</tr>
<tr>
<td>[[FunctionObject]]</td>
<td>Object</td>
<td>The function object whose invocation caused this Environment Record to be created.</td>
</tr>
<tr>
<td>[[NewTarget]]</td>
<td>Object</td>
<td>If this Environment Record was created by the [[Construct]] internal method, [[NewTarget]] is the value of the [[Construct]] newTarget parameter. Otherwise, its value is undefined.</td>
</tr>
<tr>
<td></td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BindThisValue(V)</td>
<td>Set the [[ThisValue]] and record that it has been initialized.</td>
</tr>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this Environment Record's this binding. Throws a ReferenceError if the this binding has not been initialized.</td>
</tr>
<tr>
<td>GetSuperBase()</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

### 9.1.4 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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[ObjectRecord]]</code></td>
<td>Environment Record</td>
<td>Binding object is the global object. It contains global built-in bindings as well as <code>FunctionDeclaration</code>, <code>GeneratorDeclaration</code>, <code>AsyncFunctionDeclaration</code>, <code>AsyncGeneratorDeclaration</code>, and <code>VariableDeclaration</code> bindings in global code for the associated realm.</td>
</tr>
<tr>
<td><code>[[GlobalThisValue]]</code></td>
<td>Object</td>
<td>The value returned by <code>this</code> in global scope. Hosts may provide any ECMAScript Object value.</td>
</tr>
<tr>
<td><code>[[DeclarativeRecord]]</code></td>
<td>Environment Record</td>
<td>Contains bindings for all declarations in global code for the associated realm code except for <code>FunctionDeclaration</code>, <code>GeneratorDeclaration</code>, <code>AsyncFunctionDeclaration</code>, <code>AsyncGeneratorDeclaration</code>, and <code>VariableDeclaration</code> bindings.</td>
</tr>
<tr>
<td><code>[[VarNames]]</code></td>
<td>List of String</td>
<td>The string names bound by <code>FunctionDeclaration</code>, <code>GeneratorDeclaration</code>, <code>AsyncFunctionDeclaration</code>, <code>AsyncGeneratorDeclaration</code>, and <code>VariableDeclaration</code> declarations in global code for the associated realm.</td>
</tr>
</tbody>
</table>
### Table 21: Additional Methods of Global Environment Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this Environment Record’s <code>this</code> binding.</td>
</tr>
<tr>
<td>HasVarDeclaration (N)</td>
<td>Determines if the argument identifier has a binding in this Environment Record that was created using a VariableDeclaration, FunctionDeclaration, GeneratorDeclaration, AsyncFunctionDeclaration, or AsyncGeneratorDeclaration.</td>
</tr>
<tr>
<td>HasLexicalDeclaration (N)</td>
<td>Determines if the argument identifier has a binding in this Environment Record that was created using a lexical declaration such as a LexicalDeclaration or a ClassDeclaration.</td>
</tr>
<tr>
<td>HasRestrictedGlobalProperty (N)</td>
<td>Determines if the argument is the name of a global object property that may not be shadowed by a global lexical binding.</td>
</tr>
<tr>
<td>CanDeclareGlobalVar (N)</td>
<td>Determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument N.</td>
</tr>
<tr>
<td>CanDeclareGlobalFunction (N)</td>
<td>Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument N.</td>
</tr>
<tr>
<td>CreateGlobalVarBinding(N, D)</td>
<td>Used to create and initialize to <code>undefined</code> a global <code>var</code> 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 <code>var</code>. The String value N is the bound name. If D is true the binding may be deleted. Logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows var declarations to receive special treatment.</td>
</tr>
<tr>
<td>CreateGlobalFunctionBinding(N, V, D)</td>
<td>Create and initialize a global <code>function</code> 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 <code>function</code>. The String value N is the bound name. V is the initialization value. If the Boolean argument D is true the binding may be deleted. Logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows function declarations to receive special treatment.</td>
</tr>
</tbody>
</table>

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

### 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 $envRec$.[$ObjectRecord$].

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.

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.

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
3. Assert: If the binding exists, it must be in the object Environment Record.
4. Let $ObjRec$ be $envRec$.[$ObjectRecord$].

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
3. Let `ObjRec` be `envRec`.[[ObjectRecord]].

### 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
3. Let `ObjRec` be `envRec`.[[ObjectRecord]].

### 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.

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).
5. If hasProperty is false and extensible is true, then
6. Let varDeclaredNames be envRec.[[VarNames]].
7. If varDeclaredNames does not contain N, then
a. Append \( N \) to \( \text{varDeclaredNames} \).
8. Return \text{NormalCompletion}(\emptyset).

9.1.1.18 \text{CreateGlobalFunctionBinding} (N, V, D)

The \text{CreateGlobalFunctionBinding} concrete method of a global Environment Record \( \text{envRec} \) 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 \( \text{ObjRec} \) be \( \text{envRec}.[[\text{ObjectRecord}]] \).
2. Let \( \text{globalObject} \) be the binding object for \( \text{ObjRec} \).
3. Let \( \text{existingProp} \) be \( ? \text{globalObject}.[[\text{GetOwnProperty}]](N) \).
4. If \( \text{existingProp} \) is \text{undefined} or \( \text{existingProp}.[[\text{Configurable}]] \) is \text{true}, then
   a. Let \( \text{desc} \) be the PropertyDescriptor \{ [[Value]]: \( V \), [[Writable]]: \text{true}, [[Enumerable]]: \text{true}, [[Configurable]]: \( D \) \}.
5. Else,
   a. Let \( \text{desc} \) be the PropertyDescriptor \{ [[Value]]: \( V \) \}.
6. Perform \? \text{DefinePropertyOrThrow}(\text{globalObject}, N, \text{desc}).
7. Perform \? \text{Set}(\text{globalObject}, N, V, \text{false}).
8. Let \( \text{varDeclaredNames} \) be \( \text{envRec}.[[\text{VarNames}]] \).
9. If \( \text{varDeclaredNames} \) does not contain \( N \), then
   a. Append \( N \) to \( \text{varDeclaredNames} \).
10. Return \text{NormalCompletion}(\emptyset).

\textbf{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 \text{InitializeBinding} concrete method would do and if \( \text{globalObject} \) is a Proxy will produce the same sequence of Proxy trap calls.

9.1.15 \text{Module Environment Records}

A \text{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 \text{GetBindingValue}, \text{DeleteBinding}, \text{HasThisBinding} and \text{GetThisBinding}. In addition, module Environment Records support the methods listed in Table 22:
### Table 22: Additional Methods of Module Environment Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateImportBinding(N, M, N2)</td>
<td>Create an immutable indirect binding in a module Environment Record. The String value ( N ) is the text of the bound name. ( M ) is a Module Record, and ( N2 ) is a binding that exists in ( M )’s module Environment Record.</td>
</tr>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this Environment Record’s this binding.</td>
</tr>
</tbody>
</table>

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.
NOTE

Module Environment Records always provide a **this** binding.

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 \( env.\text{[[OuterEnv]]} \) 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 \( env.\text{[[OuterEnv]]} \) 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: \( \text{Type}(newTarget) \) is Undefined or Object.
3. Let \( env \) be a new function Environment Record containing no bindings.
4. Set \( env.\text{[[FunctionObject]]} \) to \( F \).
5. If \( F.\text{[[ThisMode]]} \) is lexical, set \( env.\text{[[ThisBindingStatus]]} \) to lexical.
6. Else, set \( env.\text{[[ThisBindingStatus]]} \) to uninitialized.
7. Set \( env.\text{[[NewTarget]]} \) to \( newTarget \).
8. Set \( env.\text{[[OuterEnv]]} \) to \( F.\text{[[Environment]]} \).
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 \( env.\text{[[ObjectRecord]]} \) to \( objRec \).
5. Set \( env.\text{[[GlobalThisValue]]} \) to \( thisValue \).
6. Set \( env.\text{[[DeclarativeRecord]]} \) to \( dclRec \).
7. Set \( env.\text{[[VarNames]]} \) to a new empty List.
8. Set \( env.\text{[[OuterEnv]]} \) 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 \( env.\{\text{[OuterEnv]}\} \) 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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Intrinsics]]</td>
<td>Record whose field names are intrinsic keys and whose values are objects</td>
<td>The intrinsic values used by code associated with this realm</td>
</tr>
<tr>
<td>[[GlobalObject]]</td>
<td>Object</td>
<td>The global object for this realm</td>
</tr>
<tr>
<td>[[GlobalEnv]]</td>
<td>global Environment Record</td>
<td>The global environment for this realm</td>
</tr>
<tr>
<td>[[TemplateMap]]</td>
<td>A List of Record {</td>
<td>Template objects are canonicalized separately for each realm using its Realm Record’s [[TemplateMap]]. Each [[Site]] value is a Parse Node that is a TemplateLiteral. The associated [[Array]] value is the corresponding template object that is passed to a tag function.</td>
</tr>
<tr>
<td></td>
<td>[[Site]: Parse Node,</td>
<td>NOTE Once a Parse Node becomes unreachable, the corresponding [[Array]] is also unreachable, and it would be unobservable if an implementation removed the pair from the [[TemplateMap]] list.</td>
</tr>
<tr>
<td></td>
<td>[[Array]: Object }</td>
<td></td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is undefined</td>
<td>Field reserved for use by hosts that need to associate additional information with a Realm Record.</td>
</tr>
</tbody>
</table>

### 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 \( realmRec.\{\text{[GlobalObject]}\} \) to undefined.
4. Set \( realmRec.\{\text{[GlobalEnv]}\} \) to undefined.
5. Set \( realmRec.\{\text{[TemplateMap]}\} \) to a new empty List.
6. Return \( realmRec \).
The abstract operation CreateIntrinsics takes argument `realmRec`. It performs the following steps when called:

1. Let `intrinsics` be a new `Record`.
2. Set `realmRec.[[Intrinsics]]` to `intrinsics`.
3. Set fields of `intrinsics` 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 intrinsics and their properties must be ordered to avoid any dependencies upon objects that have not yet been created.
4. Perform `AddRestrictedFunctionProperties(intrinsics.[[Function.prototype]], realmRec)`.
5. Return `intrinsics`.

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 `intrinsics` be `realmRec.[[Intrinsics]]`.
   b. Set `globalObj` to `! OrdinaryObjectCreate(intrinsics.[[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`.

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`. 

207
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>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>code evaluation</td>
<td>Any state needed to perform, suspend, and resume evaluation of the code</td>
</tr>
<tr>
<td>state</td>
<td>associated with this execution context.</td>
</tr>
<tr>
<td>Function</td>
<td>If this execution context is evaluating the code of a function object,</td>
</tr>
<tr>
<td></td>
<td>then the value of this component is that function object. If the context</td>
</tr>
<tr>
<td></td>
<td>is evaluating the code of a Script or Module, the value is null.</td>
</tr>
<tr>
<td>Realm</td>
<td>The Realm Record from which associated code accesses ECMAScript resources.</td>
</tr>
<tr>
<td>ScriptOrModule</td>
<td>The Module Record or Script Record from which associated code originates.</td>
</tr>
<tr>
<td></td>
<td>If there is no originating script or module, as is the case for the</td>
</tr>
<tr>
<td></td>
<td>original execution context created in InitializeHostDefinedRealm, the</td>
</tr>
<tr>
<td></td>
<td>value is null.</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation. It is impossible for 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 \textit{strict} be \texttt{true}; else let \textit{strict} be \texttt{false}.

4. Return \texttt{? GetIdentifierReference(env, name, strict)}.

\begin{note}

The result of ResolveBinding is always a Reference Record whose [[ReferencedName]] field is \textit{name}.

\end{note}

### 9.3.3 GetThisEnvironment ()

The abstract operation GetThisEnvironment takes no arguments. It finds the Environment Record that currently supplies the binding of the keyword \texttt{this}. It performs the following steps when called:

1. Let \textit{env} be the running execution context's LexicalEnvironment.
2. Repeat,
   a. Let \textit{exists} be \textit{env}.HasThisBinding().
   b. If \textit{exists} is \texttt{true}, return \textit{env}.
   c. Let \textit{outer} be \textit{env}.[[OuterEnv]].
   d. Assert: \textit{outer} is not \texttt{null}.
   e. Set \textit{env} to \textit{outer}.

\begin{note}

The loop in step 2 will always terminate because the list of environments always ends with the global environment which has a \texttt{this} binding.

\end{note}

### 9.3.4 ResolveThisBinding ()

The abstract operation ResolveThisBinding takes no arguments. It determines the binding of the keyword \texttt{this} using the LexicalEnvironment of the running execution context. It performs the following steps when called:

1. Let \textit{envRec} be GetThisEnvironment().
2. Return \texttt{? 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 \textit{envRec} be GetThisEnvironment().
2. Assert: \textit{envRec} has a [[NewTarget]] field.
3. Return \textit{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 \textit{currentRealm} be the current Realm Record.
2. Return \textit{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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Callback]]</td>
<td>A function object</td>
<td>The function to invoke when the Job is invoked.</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is empty</td>
<td>Field reserved for use by hosts.</td>
</tr>
</tbody>
</table>

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.

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).
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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[LittleEndian]]</code></td>
<td>Boolean</td>
<td>The default value computed for the <code>isLittleEndian</code> parameter when it is needed by the algorithms <code>GetValueFromBuffer</code> and <code>SetValueInBuffer</code>. 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.</td>
</tr>
<tr>
<td><code>[[CanBlock]]</code></td>
<td>Boolean</td>
<td>Determines whether the agent can block or not.</td>
</tr>
<tr>
<td><code>[[Signifier]]</code></td>
<td>Any globally-unique value</td>
<td>Uniquely identifies the agent within its agent cluster.</td>
</tr>
<tr>
<td><code>[[IsLockFree1]]</code></td>
<td>Boolean</td>
<td><code>true</code> if atomic operations on one-byte values are lock-free, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>[[IsLockFree2]]</code></td>
<td>Boolean</td>
<td><code>true</code> if atomic operations on two-byte values are lock-free, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>[[IsLockFree8]]</code></td>
<td>Boolean</td>
<td><code>true</code> if atomic operations on eight-byte values are lock-free, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>[[CandidateExecution]]</code></td>
<td>A candidate execution Record</td>
<td>See the memory model.</td>
</tr>
<tr>
<td><code>[[KeptAlive]]</code></td>
<td>List of objects</td>
<td>Initially a new empty List, representing the list of objects to be kept alive until the end of the current Job</td>
</tr>
</tbody>
</table>

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`. 
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 $S$.

Colloquially, we say that an individual object is live if every set of objects containing it is live.

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

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 4

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 5

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.

NOTE 6

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 $ref.\text{[[WeakRefTarget]]}$ is $obj$, do
      i. Set $ref.\text{[[WeakRefTarget]]}$ to empty.
b. For each FinalizationRegistry \( f_g \) such that \( f_g.\text{[[Cells]]} \) contains a Record \( \text{cell} \) such that \( \text{cell}.\text{[[WeakRefTarget]]} \) is \( \text{obj} \), do
   i. Set \( \text{cell}.\text{[[WeakRefTarget]]} \) to empty.
   ii. Optionally, perform \HostEnqueueFinalizationRegistryCleanupJob\( (f_g) \).

c. For each WeakMap \( \text{map} \) such that \( \text{map}.\text{[[WeakMapData]]} \) contains a Record \( \text{r} \) such that \( \text{r}.\text{[[Key]]} \) is \( \text{obj} \), do
   i. Set \( \text{r}.\text{[[Key]]} \) to empty.
   ii. Set \( \text{r}.\text{[[Value]]} \) to empty.

d. For each WeakSet \( \text{set} \) such that \( \text{set}.\text{[[WeakSetData]]} \) contains \( \text{obj} \), do
   i. Replace the element of \( \text{set}.\text{[[WeakSetData]]} \) whose value is \( \text{obj} \) 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 \( \text{[[WeakRefTarget]]} \) 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 \( \text{obj} \) without emptying out other WeakRefs that, if not emptied, could result in an execution that observes the Object value of \( \text{obj} \).

9.9.4  Host Hooks

9.9.4.1  HostEnqueueFinalizationRegistryCleanupJob ( \( \text{finalizationRegistry} \))

The abstract operation HostEnqueueFinalizationRegistryCleanupJob takes argument \( \text{finalizationRegistry} \) (a FinalizationRegistry). HostEnqueueFinalizationRegistryCleanupJob is an implementation-defined abstract operation that is expected to call CleanupFinalizationRegistry(\( \text{finalizationRegistry} \)) 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 \( \text{agentRecord} \) be the surrounding agent’s Agent Record.
2. Set \( \text{agentRecord}.[[\text{KeptAlive}]] \) to a new empty List.

### 9.11 AddToKeptObjects (\( \text{object} \))

The abstract operation AddToKeptObjects takes argument \( \text{object} \) (an Object). It performs the following steps when called:

1. Let \( \text{agentRecord} \) be the surrounding agent’s Agent Record.
2. Append \( \text{object} \) to \( \text{agentRecord}.[[\text{KeptAlive}]] \).

**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 (\( \text{finalizationRegistry} \))

The abstract operation CleanupFinalizationRegistry takes argument \( \text{finalizationRegistry} \) (a FinalizationRegistry). It performs the following steps when called:

1. Assert: \( \text{finalizationRegistry} \) has [[Cells]] and [[CleanupCallback]] internal slots.
2. Let \( \text{callback} \) be \( \text{finalizationRegistry}.[[\text{CleanupCallback}]] \).
3. While \( \text{finalizationRegistry}.[[\text{Cells}]] \) contains a Record \( \text{cell} \) such that \( \text{cell}.[[\text{WeakRefTarget}]] \) is empty, an implementation may perform the following steps:
   a. Choose any such \( \text{cell} \).
   b. Remove \( \text{cell} \) from \( \text{finalizationRegistry}.[[\text{Cells}]] \).
   c. Perform \( \text{? Call}(\text{callback}, \text{undefined}, \text{« cell.[[HeldValue]] »}) \).
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:


#### 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.
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 $\text{true}$.

**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 $\text{[[GetPrototypeOf]]}$ and $\text{[[SetPrototypeOf]]}$.

### 10.1.3 $\text{[[IsExtensible]]}$

The $\text{[[IsExtensible]]}$ internal method of an ordinary object $O$ takes no arguments. It performs the following steps when called:

1. Return $\text{! OrdinaryIsExtensible}(O)$.

#### 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 $\text{[[PreventExtensions]]}$

The $\text{[[PreventExtensions]]}$ internal method of an ordinary object $O$ takes no arguments. It performs the following steps when called:

1. Return $\text{! OrdinaryPreventExtensions}(O)$.

#### 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 $\text{false}$.
2. Return $\text{true}$.

### 10.1.5 $\text{[[GetOwnProperty]]}$

The $\text{[[GetOwnProperty]]}$ internal method of an ordinary object $O$ takes argument $P$ (a property key). It performs the following steps when called:

1. Return $\text{! OrdinaryGetOwnProperty}(O, P)$.

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

\textbf{10.1.6} \textit{[[DefineOwnProperty]] (} \( P, \text{Desc} \) \textbf{)}

The \textit{[[DefineOwnProperty]]} internal method of an ordinary object \( O \) takes arguments \( P \) (a property key) and \( \text{Desc} \) (a \textit{Property Descriptor}). It performs the following steps when called:

1. Return \(? \text{OrdinaryDefineOwnProperty}(O, P, \text{Desc}).\)

\textbf{10.1.6.1} \textbf{OrdinaryDefineOwnProperty (} \( O, P, \text{Desc} \) \textbf{)}

The abstract operation \textit{OrdinaryDefineOwnProperty} takes arguments \( O \) (an Object), \( P \) (a property key), and \( \text{Desc} \) (a \textit{Property Descriptor}). It performs the following steps when called:

1. Let \textit{current} be \(? O.\[[\text{OwnProperty}]](P)\).
2. Let \textit{extensible} be \(? \text{IsExtensible}(O)\).
3. Return \textbf{ValidateAndApplyPropertyDescriptor}(O, \textit{extensible}, \text{Desc}, \textit{current}).

\textbf{10.1.6.2} \textbf{IsCompatiblePropertyDescriptor (} \( \text{Extensible}, \text{Desc}, \text{Current} \) \textbf{)}

The abstract operation \textit{IsCompatiblePropertyDescriptor} takes arguments \( \text{Extensible} \) (a Boolean), \( \text{Desc} \) (a \textit{Property Descriptor}), and \( \text{Current} \) (a \textit{Property Descriptor}). It performs the following steps when called:

1. Return \textbf{ValidateAndApplyPropertyDescriptor}(\textit{undefined}, \textit{undefined}, \text{Extensible}, \text{Desc}, \text{Current}).

\textbf{10.1.6.3} \textbf{ValidateAndApplyPropertyDescriptor (} \( O, P, \text{extensible}, \text{Desc}, \text{current} \) \textbf{)}

The abstract operation \textit{ValidateAndApplyPropertyDescriptor} takes arguments \( O \) (an Object or \textit{undefined}), \( P \) (a property key), \( \text{extensible} \) (a Boolean), \( \text{Desc} \) (a \textit{Property Descriptor}), and \( \text{current} \) (a \textit{Property Descriptor}). It performs the following steps when called:

\begin{itemize}
  \item \textbf{NOTE} \quad If \textit{undefined} is passed as \( O \), only validation is performed and no object updates are performed.
  \begin{enumerate}
    \item \textbf{Assert}: If \( O \) is not \textit{undefined}, then \textit{IsPropertyKey}(P) is \textbf{true}.
    \item If \textit{current} is \textit{undefined}, then
      \begin{enumerate}
        \item If \textit{extensible} is \textit{false}, return \textbf{false}.
      \end{enumerate}
  \end{enumerate}
\end{itemize}
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.
The [[HasProperty]] internal method of an ordinary object \(O\) takes argument \(P\) (a property key). It performs the following steps when called:

1. Return ? \(\text{OrdinaryHasProperty}(O, P)\).

### 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: \(\text{IsPropertyKey}(P)\) is true.
2. Let \(has\text{OwnProperty}\) be \(? O.\text{[[GetOwnProperty]]}(P)\).
3. If \(has\text{OwnProperty}\) is not undefined, return true.
4. Let \(parent\) be \(? O.\text{[[GetPrototypeOf]]}\).
5. If \(parent\) is not null, then
   a. Return ? \(parent.\text{[[HasProperty]]}(P)\).
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 ? \(\text{OrdinaryGet}(O, P, Receiver)\).

#### 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: \(\text{IsPropertyKey}(P)\) is true.
2. Let \(desc\) be \(? O.\text{[[GetOwnProperty]]}(P)\).
3. If \(desc\) is undefined, then
   a. Let \(parent\) be \(? O.\text{[[GetPrototypeOf]]}\).
   b. If \(parent\) is null, return undefined.
   c. Return ? \(parent.\text{[[Get]]}(P, Receiver)\).
4. If \(\text{IsDataDescriptor}(desc)\) is true, return \(desc.\text{[[Value]]}\).
5. Assert: \(\text{IsAccessorDescriptor}(desc)\) is true.
6. Let \(getter\) be \(desc.\text{[[Get]]}\).
7. If \(getter\) is undefined, return undefined.
8. Return ? \(\text{Call}(getter, Receiver)\).

### 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 ? \(\text{OrdinarySet}(O, P, V, Receiver)\).
10.1.9.1 OrdinarySet ( \( O, P, V, \text{Receiver} \) )

The abstract operation OrdinarySet takes arguments \( O \) (an Object), \( P \) (a property key), \( V \) (an ECMAScript language value), and \( \text{Receiver} \) (an ECMAScript language value). It performs the following steps when called:

1. Assert: IsPropertyKey(\( P \)) is true.
2. Let \( \text{ownDesc} \) be ? \( O.\text{[[GetOwnProperty]]}(P) \).
3. Return OrdinarySetWithOwnDescriptor(\( O, P, V, \text{Receiver}, \text{ownDesc} \)).

10.1.9.2 OrdinarySetWithOwnDescriptor ( \( O, P, V, \text{Receiver}, \text{ownDesc} \) )

The abstract operation OrdinarySetWithOwnDescriptor takes arguments \( O \) (an Object), \( P \) (a property key), \( V \) (an ECMAScript language value), \( \text{Receiver} \) (an ECMAScript language value), and \( \text{ownDesc} \) (a Property Descriptor or undefined). It performs the following steps when called:

1. Assert: IsPropertyKey(\( P \)) is true.
2. If \( \text{ownDesc} \) is undefined, then
   a. Let \( \text{parent} \) be ? \( O.\text{[[GetPrototypeOf]]}() \).
   b. If \( \text{parent} \) is not null, then
      i. Return ? \( \text{parent}.\text{[[Set]]}(P, V, \text{Receiver}) \).
   c. Else,
      i. Set \( \text{ownDesc} \) to the PropertyDescriptor { \( \text{[[Value]]}: \text{undefined}, \text{[[Writable]]}: \text{true}, \text{[[Enumerable]]}: \text{true}, \text{[[Configurable]]}: \text{true} \).}
3. If IsDataDescriptor(\( \text{ownDesc} \)) is true, then
   a. If \( \text{ownDesc}.\text{[[Writable]]} \) is false, return false.
   b. If Type(\( \text{Receiver} \)) is not Object, return false.
   c. Let \( \text{existingDescriptor} \) be ? \( \text{Receiver}.\text{[[GetOwnProperty]]}(P) \).
   d. If \( \text{existingDescriptor} \) is not undefined, then
      i. If IsAccessorDescriptor(\( \text{existingDescriptor} \)) is true, return false.
      ii. If \( \text{existingDescriptor}.\text{[[Writable]]} \) is false, return false.
      iii. Let \( \text{valueDesc} \) be the PropertyDescriptor { \( \text{[[Value]]}: V \).}
      iv. Return ? \( \text{Receiver}.\text{[[DefineOwnProperty]]}(P, \text{valueDesc}) \).
   e. Else,
      i. Assert: \( \text{Receiver} \) does not currently have a property \( P \).
      ii. Return ? CreateDataProperty(\( \text{Receiver}, P, V \)).
4. Assert: IsAccessorDescriptor(\( \text{ownDesc} \)) is true.
5. Let \( \text{setter} \) be \( \text{ownDesc}.\text{[[Set]]} \).
6. If \( \text{setter} \) is undefined, return false.
7. Perform ? Call(\( \text{setter}, \text{Receiver}, \langle V \rangle \)).
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: $\text{IsPropertyKey}(P)$ is $\text{true}$.
2. Let $\text{desc}$ be $O.\text{[[GetOwnProperty]]}(P)$.
3. If $\text{desc}$ is $\text{undefined}$, return $\text{true}$.
4. If $\text{desc.}[\text{Configurable}]$ is $\text{true}$, then
   a. Remove the own property with name $P$ from $O$.
   b. Return $\text{true}$.
5. Return $\text{false}$.

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 $\text{keys}$ 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 $\text{keys}$.
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 $\text{keys}$.
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 $\text{keys}$.
5. Return $\text{keys}$.

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 $\text{internalSlotsList}$ be « [[Prototype]], [[Extensible]] ».
2. If $additionalInternalSlotsList$ is present, append each of its elements to $\text{internalSlotsList}$.
3. Let $O$ be ! MakeBasicObject($\text{internalSlotsList}$).
4. Set $O.\text{[[Prototype]]}$ to $\text{proto}$.
5. Return $O$. 

228
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).

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.

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.

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.
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>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Environment]]</td>
<td>Environment Record</td>
<td>The Environment Record that the function was closed over. Used as the outer environment when evaluating the code of the function.</td>
</tr>
<tr>
<td>[[FormalParameters]]</td>
<td>Parse Node</td>
<td>The root parse node of the source text that defines the function's formal parameter list.</td>
</tr>
<tr>
<td>[[ECMAScriptCode]]</td>
<td>Parse Node</td>
<td>The root parse node of the source text that defines the function's body.</td>
</tr>
<tr>
<td>[[ConstructorKind]]</td>
<td>base</td>
<td>derived</td>
</tr>
<tr>
<td>[[Realm]]</td>
<td>Realm Record</td>
<td>The realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.</td>
</tr>
<tr>
<td>[[ScriptOrModule]]</td>
<td>Script Record or Module Record</td>
<td>The script or module in which the function was created.</td>
</tr>
<tr>
<td>[[ThisMode]]</td>
<td>lexical</td>
<td>strict</td>
</tr>
<tr>
<td>[[Strict]]</td>
<td>Boolean</td>
<td>true if this is a strict function, false if this is a non-strict function.</td>
</tr>
<tr>
<td>[[HomeObject]]</td>
<td>Object</td>
<td>If the function uses super, this is the object whose [[GetPrototypeOf]] provides the object where super property lookups begin.</td>
</tr>
<tr>
<td>[[SourceText]]</td>
<td>sequence of Unicode code points</td>
<td>The source text that defines the function.</td>
</tr>
<tr>
<td>[[IsClassConstructor]]</td>
<td>Boolean</td>
<td>Indicates whether the function is a class constructor. (If true, invoking the function’s [[Call]] will immediately throw a TypeError exception.)</td>
</tr>
</tbody>
</table>
All ECMAScript function objects have the [[Call]] internal method defined here. ECMAScript functions that are also constructors in addition have the [[Construct]] internal method.

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 calleeContext is not already suspended, suspend calleeContext.
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.
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 \( localEnv.\text{BindThisValue}(thisValue) \).

### 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 \( argumentsList \).
1. Return \texttt{EvaluateAsyncConciseBody} of \texttt{AsyncConciseBody} with arguments \texttt{functionObject} and \texttt{argumentsList}.

10.2.1.4 \texttt{OrdinaryCallEvaluateBody} ( $F$, \texttt{argumentsList} )

The abstract operation \texttt{OrdinaryCallEvaluateBody} takes arguments $F$ (a function object) and \texttt{argumentsList} (a List). It performs the following steps when called:

1. Return the result of \texttt{EvaluateBody} of the parsed code that is $F[[\text{ECMAScriptCode}]]$ passing $F$ and \texttt{argumentsList} as the arguments.

10.2.2 \texttt{[[Construct]]} ( \texttt{argumentsList}, \texttt{newTarget} )

The \texttt{[[Construct]]} internal method of an ECMAScript function object $F$ takes arguments \texttt{argumentsList} (a List of ECMAScript language values) and \texttt{newTarget} (a constructor). It performs the following steps when called:

1. \texttt{Assert}: $F$ is an ECMAScript function object.
2. \texttt{Assert}: \texttt{Type(newTarget)} is Object.
3. Let \texttt{calleeContext} be the running execution context.
4. Let \texttt{kind} be $F[[\text{ConstructorKind}]]$.
5. If \texttt{kind} is \texttt{base}, then
   a. Let \texttt{thisArgument} be ? \texttt{OrdinaryCreateFromConstructor(newTarget, "%Object.prototype").}
6. Let \texttt{calleeContext} be \texttt{PrepareForOrdinaryCall(F, newTarget)}.
7. \texttt{Assert}: \texttt{calleeContext} is now the running execution context.
8. If \texttt{kind} is \texttt{base}, perform \texttt{OrdinaryCallBindThis(F, calleeContext, thisArgument)}.
9. Let \texttt{constructorEnv} be the LexicalEnvironment of \texttt{calleeContext}.
10. Let \texttt{result} be \texttt{OrdinaryCallEvaluateBody(F, argumentsList)}.
11. Remove \texttt{calleeContext} from the execution context stack and restore \texttt{calleeContext} as the running execution context.
12. If \texttt{result}.[[Type]] is \texttt{return}, then
   a. If \texttt{Type(result}.[[Value]]) is Object, return \texttt{NormalCompletion(result}.[[Value]])
   b. If \texttt{kind} is \texttt{base}, return \texttt{NormalCompletion(thisArgument)}.
   c. If \texttt{result}.[[Value]] is not \texttt{undefined}, throw a \texttt{TypeError} exception.
13. Else, \texttt{ReturnIfAbrupt(result)}.
14. Return ? \texttt{constructorEnv}.GetThisBinding().

10.2.3 \texttt{OrdinaryFunctionCreate} ( \texttt{functionPrototype}, \texttt{sourceText}, \texttt{ParameterList}, \texttt{Body}, \texttt{thisMode}, \texttt{Scope} )

The abstract operation \texttt{OrdinaryFunctionCreate} takes arguments \texttt{functionPrototype} (an Object), \texttt{sourceText} (a sequence of Unicode code points), \texttt{ParameterList} (a Parse Node), \texttt{Body} (a Parse Node), \texttt{thisMode} (either lexical-this or non-lexical-this), and \texttt{Scope} (an Environment Record). \texttt{sourceText} is the source text of the syntactic definition of the function to be created. It performs the following steps when called:

1. \texttt{Assert}: \texttt{Type(functionPrototype)} is Object.
2. Let \texttt{internalSlotsList} be the internal slots listed in Table 29.
3. Let $F$ be ! \texttt{OrdinaryObjectCreate(functionPrototype, internalSlotsList)}.
4. Set $F.[[\text{Call}]]$ to the definition specified in 10.2.1.
5. Set $F.[[\text{SourceText}]]$ to \texttt{sourceText}.
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 $false$.
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 $false$.
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 $!\text{SetFunctionLength}(F, len)$.

### 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 $!\text{DefinePropertyOrThrow}(F, \"caller\", \{ [[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: true \})$.
4. Return $!\text{DefinePropertyOrThrow}(F, \"arguments\", \{ [[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: true \})$.

#### 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: IsConstructor($F$) is false.
   b. Assert: $F$ is an extensible object that does not have a "prototype" own property.
   c. Set $F$.[[Construct]] to the definition specified in 10.2.2.

3. Set $F$.[[ConstructorKind]] to base.

4. If writablePrototype is not present, set writablePrototype to true.

5. If prototype is not present, then
   a. Set prototype to ! OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! DefinePropertyOrThrow(prototype, "constructor", PropertyDescriptor { [[Value]]: $F$, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: true }).

6. Perform ! DefinePropertyOrThrow($F$, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: false }).

7. Return NormalCompletion(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: $F$.[[IsClassConstructor]] is false.
3. Set $F$.[[IsClassConstructor]] to true.
4. Return NormalCompletion(undefined).

10.2.7 MakeMethod ( $F$, 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: Type(homeObject) is Object.
3. Set $F$.[[HomeObject]] to homeObject.
4. Return NormalCompletion(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: Type(name) is either Symbol or String.
3. Assert: If prefix is present, then Type(prefix) is String.
4. If Type(name) is Symbol, then
   a. Let description be name's [[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 [[InitialName]] internal slot, then
   a. Set $F$.[[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 +∞). 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]]: ℓ(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
   d. Else,
   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 \( \text{lexEnv}.\text{CreateImmutableBinding}(dn, \text{true}) \).

ii. Else,  
1. Perform \( \text{lexEnv}.\text{CreateMutableBinding}(dn, \text{false}) \).

35. For each Parse Node \( f \) of \( \text{functionsToInitialize} \), do  
   a. Let \( fn \) be the sole element of the \( \text{BoundNames} \) of \( f \).
   b. Let \( fo \) be \( \text{InstantiateFunctionObject} \) of \( f \) with argument \( \text{lexEnv} \).
   c. Perform ! \( \text{varEnv}.\text{SetMutableBinding}(fn, fo, \text{false}) \).

36. Return \( \text{NormalCompletion} \)(\( \text{empty} \)).

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 \( \text{Initializers} \) may contain direct \( \text{eval} \) expressions. Any top level declarations of such \( \text{evls} \) are only visible to the \( \text{eval} \) 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 \([\text{[Prototype]}], [\text{[Extensible]}], \text{and } [\text{[Realm]}] \) internal slots.

Unless otherwise specified every built-in function object has the \( %\text{Function.prototype}% \) object as the initial value of its \([\text{[Prototype]}]\) 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 \([\text{[Call]}]\) and \([\text{[Construct]}]\) invocations of the function. However, \([\text{[Construct]}]\) invocation is not supported by all built-in functions. For each built-in function, when invoked with \([\text{[Call]}]\), the \([\text{[Call]}]\) \( \text{thisArgument} \) provides the \( \text{this} \) value, the \([\text{[Call]}]\) \( \text{argumentsList} \) provides the named parameters, and the NewTarget value is \( \text{undefined} \). When invoked with \([\text{[Construct]}]\), the \( \text{this} \) value is uninitialized, the \([\text{[Construct]}]\) \( \text{argumentsList} \) provides the named parameters, and the \([\text{[Construct]}]\) \( \text{newTarget} \) 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 \([\text{[Call]}]\) behaviour other than throwing a \( \text{TypeError} \) exception, an ECMAScript implementation of the function must be done in a manner that does not cause the function's \([\text{[IsClassConstructor]}]\) internal slot to have the value \( \text{true} \).

Built-in function objects that are not identified as constructors do not implement the \([\text{[Construct]}]\) internal method unless otherwise specified in the description of a particular function. When a built-in constructor is called as part of a \( \text{new} \) expression the \( \text{argumentsList} \) parameter of the invoked \([\text{[Construct]}]\) 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.
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

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

#### 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 \( F.[[\text{BoundTargetFunction}]] \).
2. Let \( boundThis \) be \( F.[[\text{BoundThis}]] \).
3. Let \( boundArgs \) be \( F.[[\text{BoundArguments}]] \).
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 \( F.[[\text{BoundTargetFunction}]] \).
2. Assert: IsConstructor(target) is true.
3. Let \( boundArgs \) be \( F.[[\text{BoundArguments}]] \).
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.

#### 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
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 + 1_F`.
      ii. Let `succeeded` be ` OrdinaryDefineOwnProperty(A, "length", oldLenDesc)`.
      iii. **Assert:** `succeeded` is true.
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]]: \textbackslash u1D53D(\texttt{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, « \textbackslash u1D53D(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:

k. Return true.
1. If \( \text{Desc} .[[\text{Value}]] \) is absent, then
   a. Return \( \text{OrdinaryDefineOwnProperty}(A, "length", \text{Desc}) \).

2. Let \( \text{newLenDesc} \) be a copy of \( \text{Desc} \).
3. Let \( \text{newLen} \) be \( \text{ToUint32} (\text{Desc} .[[\text{Value}]]) \).
4. Let \( \text{numberLen} \) be \( \text{ToNumber} (\text{Desc} .[[\text{Value}]]) \).
5. If \( \text{newLen} \) is not the same value as \( \text{numberLen} \), throw a \text{RangeError} exception.
6. Set \( \text{newLenDesc} .[[\text{Value}]] \) to \( \text{newLen} \).
7. Let \( \text{oldLenDesc} \) be \( \text{OrdinaryGetOwnProperty}(A, "length") \).
8. Assert: \( ! \text{IsDataDescriptor} (\text{oldLenDesc}) \) is true.
9. Assert: \( \text{oldLenDesc} .[[\text{Configurable}]] \) is false.
10. Let \( \text{oldLen} \) be \( \text{oldLenDesc} .[[\text{Value}]] \).
11. If \( \text{newLen} \geq \text{oldLen} \), then
    a. Return \( \text{OrdinaryDefineOwnProperty}(A, "length", \text{newLenDesc}) \).
12. If \( \text{oldLenDesc} .[[\text{Writable}]] \) is false, return \text{false}.
13. If \( \text{newLenDesc} .[[\text{Writable}]] \) is absent or has the value \text{true}, let \( \text{newWritable} \) be \text{true}.
14. Else,
    a. NOTE: Setting the \([\text{Writable}]\) attribute to false is deferred in case any elements cannot be deleted.
    b. Let \( \text{newWritable} \) be false.
    c. Set \( \text{newLenDesc} .[[\text{Writable}]] \) to true.
15. Let \( \text{succeeded} \) be \( \text{OrdinaryDefineOwnProperty}(A, "length", \text{newLenDesc}) \).
16. If \( \text{succeeded} \) is \text{false}, return \text{false}.
17. For each own property key \( P \) of \( A \) that is an array index, whose numeric value is greater than or equal to \( \text{newLen} \), in descending numeric index order, do
    a. Let \( \text{deleteSucceeded} \) be \( ! A .[[\text{Delete}]] (P) \).
    b. If \( \text{deleteSucceeded} \) is \text{false}, then
        i. Set \( \text{newLenDesc} .[[\text{Value}]] \) to \( ! \text{ToUint32} (P) + 1 \).
        ii. If \( \text{newWritable} \) is false, set \( \text{newLenDesc} .[[\text{Writable}]] \) to false.
        iii. Perform \( \text{OrdinaryDefineOwnProperty}(A, "length", \text{newLenDesc}) \).
        iv. Return false.
18. If \( \text{newWritable} \) is \text{false}, then
    a. Let \( \text{succeeded} \) be \( \text{OrdinaryDefineOwnProperty}(A, "length", \text{PropertyDescriptor} ( \{ [[\text{Writable}]]: \text{false} \} )) \).
    b. Assert: \( \text{succeeded} \) is true.
19. Return true.

NOTE
In steps 3 and 4, if \( \text{Desc} .[[\text{Value}]] \) is an object then its \text{valueOf} method is called twice. This is legacy behaviour that was specified with this effect starting with the 2\text{nd} 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]]: F(length), [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false })`.

### 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 `-\infty`, return `undefined`.
8. Let `str` be `S.[[StringData]]`.
9. Assert: `Type(str)` is String.
10. Let `len` be the length of `str`.
11. If `\Re(index) < 0` or `len \leq \Re(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 `\Re(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
Ordinary arguments objects also have a \([\text{ParameterMap}]\) internal slot whose value is always undefined. For ordinary argument objects the \([\text{ParameterMap}]\) internal slot is only used by \texttt{Object.prototype.toString}(20.1.3.6) to identify them as such.

**NOTE 2**

The integer-indexed data properties of an \textit{arguments exotic object} whose numeric name values are less than the number of formal parameters of the corresponding \textit{function object} initially share their values with the corresponding argument bindings in the function's \textit{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 \textit{accessor property}. If the arguments object is an \textit{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 \textit{accessor property} named "\texttt{callee}" which throws a \texttt{TypeError} exception on access. The "\texttt{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 \textit{accessor property} named "\texttt{caller}". Prior to ECMAScript 2017, this specification included the definition of a throwing "\texttt{caller}" property on ordinary arguments objects. Since implementations do not contain this extension any longer, ECMAScript 2017 dropped the requirement for a throwing "\texttt{caller}" accessor.

### 10.4.4.1  \[\text{[[GetOwnProperty]]} (\textit{P})\]

The \[\text{[[GetOwnProperty]]}\] internal method of an \textit{arguments exotic object} \textit{args} takes argument \textit{P} (a property key). It performs the following steps when called:

1. Let \textit{desc} be \text{OrdinaryGetOwnProperty}({\textit{args, P}}).
2. If \textit{desc} is \texttt{undefined}, return \textit{desc}.
3. Let \textit{map} be \textit{args}.\[\text{[[ParameterMap]]}\].
4. Let \textit{isMapped} be \! \text{HasOwnProperty}({\textit{map, P}}).
5. If \textit{isMapped} is \texttt{true}, then
   a. Set \textit{desc}.\[\text{[[Value]]}\] to \text{Get}({\textit{map, P}}).
6. Return \textit{desc}.

### 10.4.4.2  \[\text{[[DefineOwnProperty]]} (\textit{P, Desc})\]

248
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
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.


### 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]]: $f(len), [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true })`.
5. Let `index` be `0`.
6. Repeat, while `index < len`,
   a. Let `val` be `argumentsList[index]`.
   b. Perform `! CreateDataPropertyOrThrow(obj, ! ToString($f(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 })`.

### 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` as specified in 10.4.4.5.
9. Set `obj` to `%Object.prototype%`.
10. Let `map` be ` OrdinaryObjectCreate(null)`.
11. Set `obj` 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(f(index)), val)`.
   c. Set `index` to `index + 1`.
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(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 })`.
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]]`. 

251
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 \textbf{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 \textbf{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)`.


### 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)`.

The \[\text{Set}\] internal method of an \text{Integer-Indexed exotic object} \(O\) takes arguments \(P\) (a property key), \(V\) (an ECMAScript language value), and \(\text{Receiver}\) (an ECMAScript language value). It performs the following steps when called:

1. \text{Assert: IsPropertyKey}(P) is \text{true}.
2. If \(\text{Type}(P)\) is String, then
   a. Let \(\text{numericIndex}\) be \(!\ \text{CanonicalNumericIndexString}(P)\).
   b. If \(\text{numericIndex}\) is not \text{undefined}, then
      i. Perform \(?\ \text{IntegerIndexedElementSet}(O, \text{numericIndex}, V)\).
      ii. Return \text{true}.
3. Return \(?\ \text{OrdinarySet}(O, P, V, \text{Receiver})\).

The \[\text{Delete}\] internal method of an \text{Integer-Indexed exotic object} \(O\) takes arguments \(P\) (a property key). It performs the following steps when called:

1. \text{Assert: IsPropertyKey}(P) is \text{true}.
2. \text{Assert: } O \text{ is an Integer-Indexed exotic object.}
3. If \(\text{Type}(P)\) is String, then
   a. Let \(\text{numericIndex}\) be \(!\ \text{CanonicalNumericIndexString}(P)\).
   b. If \(\text{numericIndex}\) is not \text{undefined}, then
      i. If \(!\ \text{IsValidIntegerIndex}(O, \text{numericIndex})\) is \text{false}, return \text{true}; else return \text{false}.
4. Return \(?\ \text{OrdinaryDelete}(O, P)\).

The \[\text{OwnPropertyKeys}\] internal method of an \text{Integer-Indexed exotic object} \(O\) takes no arguments. It performs the following steps when called:

1. Let \(\text{keys}\) be a new empty \text{List}.
2. \text{Assert: } O \text{ is an Integer-Indexed exotic object.}
3. If \(\text{IsDetachedBuffer}(O, [\text{ViewedArrayBuffer}])\) is \text{false}, then
   a. For each integer \(i\) starting with 0 such that \(i < O[[\text{ArrayLength}]]\), in ascending order, do
      i. Add \(!\ \text{ToString}(i)\) as the last element of \(\text{keys}\).
4. For each own property key \(P\) of \(O\) such that \(\text{Type}(P)\) 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 \(\text{keys}\).
5. For each own property key \(P\) of \(O\) such that \(\text{Type}(P)\) is Symbol, in ascending chronological order of property creation, do
   a. Add \(P\) as the last element of \(\text{keys}\).
6. Return \(\text{keys}\).

The abstract operation \text{IntegerIndexedObjectCreate} takes argument \text{prototype}. It is used to specify the creation of new \text{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`, return false.
5. If `ℝ(index) < 0` or `ℝ(index) ≥ 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 `(ℝ(index) × elementSize) + 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.[[TypedArrayName]]`.

c. Let `elementSize` be the Element Size value specified in Table 60 for `arrayTypeName`.

d. Let `indexedPosition` be `(R(index) \times elementSize) + offset`.

e. Let `elementType` be the Element Type value in Table 60 for `arrayTypeName`.

f. Perform `SetValueInBuffer(O.[[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>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Module]]</code></td>
<td>Module Record</td>
<td>The Module Record whose exports this namespace exposes.</td>
</tr>
<tr>
<td><code>[[Exports]]</code></td>
<td>List of String</td>
<td>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 <code>undefined</code> as <code>comparefn</code>.</td>
</tr>
<tr>
<td><code>[[Prototype]]</code></td>
<td>Null</td>
<td>This slot always contains the value <code>null</code> (see 10.4.6.1).</td>
</tr>
</tbody>
</table>

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:


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.
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 ? $O$.[[OwnProperty]]($P$).
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
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
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
3. Let `exports` be `O. [[Exports]]`.
4. If `P` is an element of `exports`, return `false`.

258
5. Return \texttt{true}.

\section*{10.4.6.10 \texttt{OwnPropertyKeys} ()}

The \texttt{OwnPropertyKeys} internal method of a \textit{module namespace exotic object} \texttt{O} takes no arguments. It performs the following steps when called:

1. Let \texttt{exports} be a copy of \texttt{O.\[Exports\]}.
2. Let \texttt{symbolKeys} be \texttt{! OrdinaryOwnPropertyKeys(O)}.
3. Append all the entries of \texttt{symbolKeys} to the end of \texttt{exports}.
4. Return \texttt{exports}.

\section*{10.4.6.11 ModuleNamespaceCreate ( \texttt{module, exports} )}

The abstract operation ModuleNamespaceCreate takes arguments \texttt{module} and \texttt{exports}. It is used to specify the creation of new module namespace exotic objects. It performs the following steps when called:

1. \texttt{Assert: module} is a \texttt{Module Record}.
2. \texttt{Assert: module.\[Namespace\]} is \texttt{undefined}.
3. \texttt{Assert: exports} is a \texttt{List} of String values.
4. Let \texttt{internalSlotsList} be the internal slots listed in Table 31.
5. Let \texttt{M} be \texttt{! MakeBasicObject(internalSlotsList)}.
6. Set \texttt{M}'s essential internal methods to the definitions specified in 10.4.6.
7. Set \texttt{M.\[Prototype\]} to \texttt{null}.
8. Set \texttt{M.\[Module\]} to \texttt{module}.
9. Let \texttt{sortedExports} be a \texttt{List} whose elements are the elements of \texttt{exports} ordered as if an Array of the same values had been sorted using \texttt{%Array.prototype.sort%} using \texttt{undefined} as \texttt{comparefn}.
10. Set \texttt{M.\[Exports\]} to \texttt{sortedExports}.
11. Create own properties of \texttt{M} corresponding to the definitions in 28.3.
12. Set \texttt{module.\[Namespace\]} to \texttt{M}.
13. Return \texttt{M}.

\section*{10.4.7 Immutable Prototype Exotic Objects}

An \textit{immutable prototype exotic object} is an \textit{exotic object} that has a [[Prototype]] internal slot that will not change once it is initialized.

An object is an \textit{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 \textit{immutable prototype exotic object} in question.)

\section*{NOTE}

Unlike other exotic objects, there is not a dedicated creation abstract operation provided for \textit{immutable prototype exotic objects}. This is because they are only used by \texttt{%Object.prototype%} and by \texttt{host} environments, and in \texttt{host} environments, the relevant objects are potentially exotic in other ways and thus need their own dedicated creation operation.

\section*{10.4.7.1 \texttt{SetPrototypeOf} ( \texttt{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 ? \( O[[\text{GetPrototypeOf}]]() \).
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

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

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

The [[ProxyHandler]] and [[ProxyTarget]] internal slots of a proxy object are always initialized when the object is created and typically may not be modified. Some proxy objects are created in a manner that permits them to be subsequently revoked. When a proxy is revoked, its [[ProxyHandler]] and [[ProxyTarget]] internal slots are set to null causing subsequent invocations of internal methods on that proxy object to throw a TypeError exception.

Because proxy 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 \text{null}, throw a \text{TypeError} exception.
3. \text{Assert: Type}(\text{handler}) is Object.
4. Let \( target \) be \( O.\text{[ProxyTarget]} \).
5. Let \( trap \) be \text{GetMethod}(\text{handler}, "getPrototypeOf").
6. If \( trap \) is \text{undefined}, then
   a. Return \( ? \text{target.}[\text{GetPrototypeOf}]() \).
7. Let \( handlerProto \) be \text{Call}(\text{trap}, \text{handler}, « \text{target } »).
8. If \text{Type}(\text{handlerProto}) is neither Object nor Null, throw a \text{TypeError} exception.
9. Let \( extensibleTarget \) be \text{IsExtensible}(\text{target}).
10. If \( extensibleTarget \) is \text{true}, return \( handlerProto \).
11. Let \( targetProto \) be \( ? \text{target.}[\text{GetPrototypeOf}]() \).
12. If \text{SameValue}(\text{handlerProto}, \text{targetProto}) is \text{false}, throw a \text{TypeError} exception.
13. Return \( handlerProto \).

\(\text{NOTE}\) \([\text{GetPrototypeOf}]\) for proxy objects enforces the following invariants:

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

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

The \([\text{SetPrototypeOf}]\) internal method of a \text{Proxy} exotic object \( O \) takes argument \( V \) (an Object or \text{null}). It performs the following steps when called:

1. \text{Assert: Either Type}(V) is Object or Type(V) is Null.
2. Let \( \text{handler} \) be \( O.\text{[ProxyHandler]} \).
3. If \( \text{handler} \) is \text{null}, throw a \text{TypeError} exception.
4. \text{Assert: Type}(\text{handler}) is Object.
5. Let \( \text{target} \) be \( O.\text{[ProxyTarget]} \).
6. Let \( \text{trap} \) be \text{GetMethod}(\text{handler}, "setPrototypeOf").
7. If \( \text{trap} \) is \text{undefined}, then
   a. Return \( ? \text{target.}[\text{SetPrototypeOf}](V) \).
8. Let \( \text{booleanTrapResult} \) be \text{ToBoolean}(\text{Call}(\text{trap}, \text{handler}, « \text{target, V »}))
9. If \( \text{booleanTrapResult} \) is \text{false}, return \text{false}.
10. Let \( \text{extensibleTarget} \) be \text{IsExtensible}(\text{target}).
11. If \( \text{extensibleTarget} \) is \text{true}, return \text{true}.
12. Let \( \text{targetProto} \) be \( ? \text{target.}[\text{GetPrototypeOf}]() \).
13. If \text{SameValue}(V, \text{targetProto}) is \text{false}, throw a \text{TypeError} exception.
14. Return \text{true}.
NOTE \[\text{[[PreventExtensions]]}\] for proxy objects enforces the following invariants:

- The result of \[\text{[[PreventExtensions]]}\] is a Boolean value.
- \[\text{[[PreventExtensions]]}\] applied to the proxy object must return the same value as \[\text{[[PreventExtensions]]}\] applied to the proxy object's target object with the same argument.

### 10.5.3 \[\text{[[IsExtensible]]}\] ( )

The \[\text{[[IsExtensible]]}\] internal method of a \textit{Proxy exotic object} \(O\) takes no arguments. It performs the following steps when called:

1. Let \(\text{handler}\) be \(O.\text{[[ProxyHandler]]}\).
2. If \(\text{handler}\) is \text{null}, throw a \text{TypeError} exception.
3. \textbf{Assert:} \(\text{Type(}\text{handler}\text{)}\) is Object.
4. Let \(\text{target}\) be \(O.\text{[[ProxyTarget]]}\).
5. Let \(\text{trap}\) be ? \text{GetMethod}(\text{handler}, "isExtensible").
6. If \(\text{trap}\) is \text{undefined}, then
   a. Return ? \text{IsExtensible}(\text{target}).
7. Let \(\text{booleanTrapResult}\) be ! \text{ToBoolean(? Call(\text{trap}, \text{handler}, « \text{target »})}).
8. Let \(\text{targetResult}\) be ? \text{IsExtensible}(\text{target}).
9. If \(\text{SameValue(}\text{booleanTrapResult}, \text{targetResult})\) is \text{false}, throw a \text{TypeError} exception.
10. Return \(\text{booleanTrapResult}\).

### 10.5.4 \[\text{[[PreventExtensions]]}\] ( )

The \[\text{[[PreventExtensions]]}\] internal method of a \textit{Proxy exotic object} \(O\) takes no arguments. It performs the following steps when called:

1. Let \(\text{handler}\) be \(O.\text{[[ProxyHandler]]}\).
2. If \(\text{handler}\) is \text{null}, throw a \text{TypeError} exception.
3. \textbf{Assert:} \(\text{Type(}\text{handler}\text{)}\) is Object.
4. Let \(\text{target}\) be \(O.\text{[[ProxyTarget]]}\).
5. Let \(\text{trap}\) be ? \text{GetMethod}(\text{handler}, "preventExtensions").
6. If \(\text{trap}\) is \text{undefined}, then
   a. Return \(\text{target}.\text{[[PreventExtensions]]}()\).
7. Let \(\text{booleanTrapResult}\) be ! \text{ToBoolean(? Call(\text{trap}, \text{handler}, « \text{target »})}).
8. If \(\text{booleanTrapResult}\) is \text{true}, then
   a. Let \(\text{extensibleTarget}\) be ? \text{IsExtensible}(\text{target}).
   b. If \(\text{extensibleTarget}\) is \text{true}, throw a \text{TypeError} exception.
9. Return \(\text{booleanTrapResult}\).
[[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
8. Let trapResultObj be ? Call(trap, handler, « target, P »).
9. If Type(trapResultObj) is neither Object nor Undefined, throw a TypeError exception.
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: 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
8. Let $descObj$ be FromPropertyDescriptor($Desc$).
9. Let booleanTrapResult be ! ToBoolean(? Call($trap$, $handler$, « $target$, $P$, $descObj$ »)).
10. If booleanTrapResult is false, return false.
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.

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:

- The result of [[DefineOwnProperty]] 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.
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
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`.

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.

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]])`.
7. If `trap` is `undefined`, then
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]])`.
7. If `trap` is `undefined`, then
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]]`.
7. If `trap` is `undefined`, then
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 \( \text{handler} \) be \( O.\text{[[ProxyHandler]]} \).
2. If \( \text{handler} \) is \text{null}, throw a \text{TypeError} exception.
3. Assert: \text{Type(\text{handler})} is \text{Object}.
4. Let \( \text{target} \) be \( O.\text{[[ProxyTarget]]} \).
5. Let \( \text{trap} \) be \( ? \text{GetMethod(\text{handler}, "ownKeys")} \).
6. If \( \text{trap} \) is \text{undefined}, then
   a. Return \( ? \text{target.[[OwnPropertyKeys]]}() \).
7. Let \( \text{trapResultArray} \) be \( ? \text{Call(\text{trap, handler, « target »})} \).
8. Let \( \text{trapResult} \) be \( ? \text{CreateListFromArrayLike(trapResultArray, « String, Symbol »)} \).
9. If \( \text{trapResult} \) contains any duplicate entries, throw a \text{TypeError} exception.
10. Let \( \text{extensibleTarget} \) be \( ? \text{IsExtensible(\text{target})} \).
11. Let \( \text{targetKeys} \) be \( ? \text{\text{target.[[OwnPropertyKeys]]}()}. \)
12. Assert: \( \text{targetKeys} \) is a \text{List} whose elements are only String and Symbol values.
13. Assert: \( \text{targetKeys} \) contains no duplicate entries.
14. Let \( \text{targetConfigurableKeys} \) be a new empty \text{List}.
15. Let \( \text{targetNonconfigurableKeys} \) be a new empty \text{List}.
16. For each element \( \text{key} \) of \( \text{targetKeys} \), do
   a. Let \( \text{desc} \) be \( ? \text{\text{target. [[GetOwnProperty]](\text{key})}} \).
   b. If \( \text{desc} \) is not \text{undefined} and \( \text{desc. [[Configurable]]} \) is \text{false}, then
      i. Append \( \text{key} \) as an element of \( \text{targetNonconfigurableKeys} \).
   c. Else,
      i. Append \( \text{key} \) as an element of \( \text{targetConfigurableKeys} \).
17. If \( \text{extensibleTarget} \) is \text{true} and \( \text{targetNonconfigurableKeys} \) is empty, then
   a. Return \( \text{trapResult} \).
18. Let \( \text{uncheckedResultKeys} \) be a \text{List} whose elements are the elements of \( \text{trapResult} \).
19. For each element \( \text{key} \) of \( \text{targetNonconfigurableKeys} \), do
   a. If \( \text{key} \) is not an element of \( \text{uncheckedResultKeys} \), throw a \text{TypeError} exception.
   b. Remove \( \text{key} \) from \( \text{uncheckedResultKeys} \).
20. If \( \text{extensibleTarget} \) is \text{true}, return \( \text{trapResult} \).
21. For each element \( \text{key} \) of \( \text{targetConfigurableKeys} \), do
   a. If \( \text{key} \) is not an element of \( \text{uncheckedResultKeys} \), throw a \text{TypeError} exception.
   b. Remove \( \text{key} \) from \( \text{uncheckedResultKeys} \).
22. If \( \text{uncheckedResultKeys} \) is not empty, throw a \text{TypeError} exception.
23. Return \( \text{trapResult} \).

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

- The result of [[OwnPropertyKeys]] is a \text{List}.
- The returned \text{List} contains no duplicate entries.
- The Type of each result \text{List} element is either String or Symbol.
- The result \text{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 \text{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.
7. If trap is undefined, then
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 floor((\(cp\) - 0x10000) / 0x400) + 0xD800.
4. Let \(cu2\) be the code unit whose value is ((\(cp\) - 0x10000) modulo 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 ( \( \text{string, position} \) )

The abstract operation \( \text{CodePointAt} \) takes arguments \( \text{string} \) (a String) and \( \text{position} \) (a non-negative integer). It interprets \( \text{string} \) 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 \( \text{position} \). It performs the following steps when called:

1. Let \( \text{size} \) be the length of \( \text{string} \).
2. Assert: \( \text{position} \geq 0 \) and \( \text{position} < \text{size} \).
3. Let \( \text{first} \) be the code unit at index \( \text{position} \) within \( \text{string} \).
4. Let \( \text{cp} \) be the code point whose numeric value is that of \( \text{first} \).
5. If \( \text{first} \) is not a leading surrogate or trailing surrogate, then
   a. Return the Record \( \{ [[\text{CodePoint}]]: \text{cp}, [[\text{CodeUnitCount}]]: 1, [[\text{IsUnpairedSurrogate}]]: \text{false} \} \).
6. If \( \text{first} \) is a trailing surrogate or \( \text{position} + 1 = \text{size} \), then
   a. Return the Record \( \{ [[\text{CodePoint}]]: \text{cp}, [[\text{CodeUnitCount}]]: 1, [[\text{IsUnpairedSurrogate}]]: \text{true} \} \).
7. Let \( \text{second} \) be the code unit at index \( \text{position} + 1 \) within \( \text{string} \).
8. If \( \text{second} \) is not a trailing surrogate, then
   a. Return the Record \( \{ [[\text{CodePoint}]]: \text{cp}, [[\text{CodeUnitCount}]]: 1, [[\text{IsUnpairedSurrogate}]]: \text{true} \} \).
9. Set \( \text{cp} \) to \( \text{UTF16SurrogatePairToCodePoint}(\text{first}, \text{second}) \).
10. Return the Record \( \{ [[\text{CodePoint}]]: \text{cp}, [[\text{CodeUnitCount}]]: 2, [[\text{IsUnpairedSurrogate}]]: \text{false} \} \).

### 11.1.5 Static Semantics: StringToCodePoints ( \( \text{string} \) )

The abstract operation \( \text{StringToCodePoints} \) takes argument \( \text{string} \) (a String). It returns the sequence of Unicode code points that results from interpreting \( \text{string} \) as UTF-16 encoded Unicode text as described in 6.1.4. It performs the following steps when called:

1. Let \( \text{codePoints} \) be a new empty List.
2. Let \( \text{size} \) be the length of \( \text{string} \).
3. Let \( \text{position} \) be 0.
4. Repeat, while \( \text{position} < \text{size} \),
   a. Let \( \text{cp} \) be \( \text{CodePointAt}(\text{string}, \text{position}) \).
   b. Append \( \text{cp}.[[\text{CodePoint}]] \) to \( \text{codePoints} \).
   c. Set \( \text{position} \) to \( \text{position} + \text{cp}.[[\text{CodeUnitCount}]] \).
5. Return \( \text{codePoints} \).

### 11.1.6 Static Semantics: ParseText ( \( \text{sourceText, goalSymbol} \) )

The abstract operation \( \text{ParseText} \) takes arguments \( \text{sourceText} \) (a sequence of Unicode code points) and \( \text{goalSymbol} \) (a nonterminal in one of the ECMAScript grammars). It performs the following steps when called:

1. Attempt to parse \( \text{sourceText} \) using \( \text{goalSymbol} \) 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 \( \text{goalSymbol} \)) 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
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 ::
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 <ZWNBSP> 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>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Abbreviation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+200C</td>
<td>ZERO WIDTH NON-JOINER</td>
<td>&lt;ZWNJ&gt;</td>
<td>IdentifierPart</td>
</tr>
<tr>
<td>U+200D</td>
<td>ZERO WIDTH JOINER</td>
<td>&lt;ZWJ&gt;</td>
<td>IdentifierPart</td>
</tr>
<tr>
<td>U+FEFF</td>
<td>ZERO WIDTH NO-BREAK SPACE</td>
<td>&lt;ZWNBSP&gt;</td>
<td>WhiteSpace</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>U+000B</td>
<td>LINE TABULATION</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>U+000C</td>
<td>FORM FEED (FF)</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>U+0020</td>
<td>SPACE</td>
<td>&lt;SP&gt;</td>
</tr>
<tr>
<td>U+00A0</td>
<td>NO-BREAK SPACE</td>
<td>&lt;NBSP&gt;</td>
</tr>
<tr>
<td>U+FEFF</td>
<td>ZERO WIDTH NO-BREAK SPACE</td>
<td>&lt;ZWNBSP&gt;</td>
</tr>
<tr>
<td>Other category “Zs”</td>
<td>Any other Unicode “Space_Separator” code point</td>
<td>&lt;USP&gt;</td>
</tr>
</tbody>
</table>

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”).
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>
<thead>
<tr>
<th>Code Point</th>
<th>Unicode Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+000A</td>
<td>LINE FEED (LF)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>U+000D</td>
<td>CARRIAGE RETURN (CR)</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>U+2028</td>
<td>LINE SEPARATOR</td>
<td>&lt;LS&gt;</td>
</tr>
<tr>
<td>U+2029</td>
<td>PARAGRAPH SEPARATOR</td>
<td>&lt;PS&gt;</td>
</tr>
</tbody>
</table>

Only the Unicode code points in Table 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

```plaintext
WhiteSpace ::
   <TAB>
   <VT>
   <FF>
   <SP>
   <NBSP>
   <ZWNBS>
   <USP>
```

LineTerminator ::
   <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 ::
    /* 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 *
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 DivPunctuator, 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

\[
\text{IdentifierName} ::
\]

\[
\text{IdentifierStart}
\]

\[
\text{IdentifierName} \ \text{IdentifierPart}
\]

\[
\text{IdentifierStart} ::
\]

\[
\text{UnicodeIDStart}
\]

\[
\$
\]

\[
\_\ \text{UnicodeEscapeSequence}
\]

\[
\text{IdentifierPart} ::
\]

\[
\text{UnicodeIDContinue}
\]

\[
\$
\]

\[
\_\ \text{UnicodeEscapeSequence}
\]

\[
<\text{ZWNJ}>
\]

\[
<\text{ZWJ}>
\]

\[
\text{UnicodeIDStart} ::
\]

\[
\text{any Unicode code point with the Unicode property “ID_Start”}
\]

\[
\text{UnicodeIDContinue} ::
\]

\[
\text{any Unicode code point with the Unicode property “ID_Continue”}
\]

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

\[
\text{IdentifierStart} :: \_ \ \text{UnicodeEscapeSequence}
\]

- 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.
It is a Syntax Error if the SV of UnicodeEscapeSequence is none of "\", \u005b\u005d, \u005c, or UTF16EncodeCodePoint(<ZWNJ>), or 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 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 \[\text{Sep}\] ::
  DecimalDigit
  DecimalDigits \[?\text{Sep}\] DecimalDigit
  \+[?\text{Sep}\] DecimalDigits \+[?\text{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 \[\text{Sep}\] ::
  ExponentIndicator SignedInteger \[?\text{Sep}\]

ExponentIndicator :: one of
  \text{e} \text{E}'

SignedInteger \[\text{Sep}\] ::
  DecimalDigits \[?\text{Sep}\]
  + DecimalDigits \[?\text{Sep}\]
  - DecimalDigits \[?\text{Sep}\]

BinaryIntegerLiteral \[\text{Sep}\] ::
  \text{0b} BinaryDigits \[?\text{Sep}\]
  \text{0B} BinaryDigits \[?\text{Sep}\]

BinaryDigits \[\text{Sep}\] ::
  BinaryDigit
  BinaryDigits \[?\text{Sep}\] BinaryDigit
  \+[?\text{Sep}\] BinaryDigits \+[?\text{Sep}\] NumericLiteralSeparator BinaryDigit

BinaryDigit :: one of
  0 1

OctalIntegerLiteral \[\text{Sep}\] ::
  \text{0o} OctalDigits \[?\text{Sep}\]
  \text{0O} OctalDigits \[?\text{Sep}\]

OctalDigits \[\text{Sep}\] ::
  OctalDigit
  OctalDigits \[?\text{Sep}\] OctalDigit
  \+[?\text{Sep}\] OctalDigits \+[?\text{Sep}\] NumericLiteralSeparator OctalDigit

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

HexIntegerLiteral \[\text{Sep}\] ::
0x HexDigits
0X HexDigits

HexDigits ::
  HexDigit

HexDigit :: [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 × 10ⁿ), 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 × 10ⁿ, where e is the MV of ExponentPart.
- The MV of DecimalLiteral :: DecimalIntegerLiteral . DecimalDigits ExponentPart is (the MV of DecimalIntegerLiteral plus (the MV of DecimalDigits × 10ⁿ)) × 10ⁿ, 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 × 10ⁿ, 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 × 10ⁿ·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 × 10ⁿ, 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 \times 10) plus the MV of DecimalDigit.
- The MV of DecimalDigits :: DecimalDigits NumericLiteralSeparator DecimalDigit is (the MV of DecimalDigits \times 10) plus the MV of DecimalDigit.
- The MV of ExponentPart :: ExponentIndicator SignedInteger is the MV of SignedInteger.
- The MV of SignedInteger :: DecimalDigits is the MV of DecimalDigits.
- The MV of SignedInteger :: + DecimalDigits is the MV of DecimalDigits.
- The MV of SignedInteger :: - DecimalDigits is the negative of the MV of DecimalDigits.
- The MV of DecimalDigit :: 0 or of HexDigit :: 0 or of OctalDigit :: 0 or of BinaryDigit :: 0 is 0.
- The MV of DecimalDigit :: 1 or of NonZeroDigit :: 1 or of HexDigit :: 1 or of OctalDigit :: 1 or of BinaryDigit :: 1 is 1.
- The MV of DecimalDigit :: 2 or of NonZeroDigit :: 2 or of HexDigit :: 2 or of OctalDigit :: 2 is 2.
- The MV of DecimalDigit :: 3 or of NonZeroDigit :: 3 or of HexDigit :: 3 or of OctalDigit :: 3 is 3.
- The MV of DecimalDigit :: 4 or of NonZeroDigit :: 4 or of HexDigit :: 4 or of OctalDigit :: 4 is 4.
- The MV of DecimalDigit :: 5 or of NonZeroDigit :: 5 or of HexDigit :: 5 or of OctalDigit :: 5 is 5.
- The MV of DecimalDigit :: 6 or of NonZeroDigit :: 6 or of HexDigit :: 6 or of OctalDigit :: 6 is 6.
- The MV of DecimalDigit :: 7 or of NonZeroDigit :: 7 or of HexDigit :: 7 or of OctalDigit :: 7 is 7.
- The MV of DecimalDigit :: 8 or of NonZeroDigit :: 8 or of HexDigit :: 8 is 8.
- The MV of DecimalDigit :: 9 or of NonZeroDigit :: 9 or of HexDigit :: 9 is 9.
- The MV of HexDigit :: a or of HexDigit :: A is 10.
- The MV of HexDigit :: b or of HexDigit :: B is 11.
- The MV of HexDigit :: c or of HexDigit :: C is 12.
- The MV of HexDigit :: d or of HexDigit :: D is 13.
- The MV of HexDigit :: e or of HexDigit :: E is 14.
- The MV of HexDigit :: f or of HexDigit :: F is 15.
- The MV of BinaryIntegerLiteral :: 0b 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 \times 2) plus the MV of BinaryDigit.
- The MV of BinaryDigits :: BinaryDigits NumericLiteralSeparator BinaryDigit is (the MV of BinaryDigits \times 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 \times 8) plus the MV of OctalDigit.
- The MV of OctalDigits :: OctalDigits NumericLiteralSeparator OctalDigit is (the MV of OctalDigits \times 8) plus the MV of OctalDigit.
- The MV of HexIntegerLiteral :: 0x HexDigits is the MV of HexDigits.
- The MV of HexIntegerLiteral :: 0X HexDigits is the MV of HexDigits.
- The MV of HexDigits :: HexDigit is the MV of HexDigit.
- The MV of HexDigits :: HexDigits HexDigit is (the MV of HexDigits \times 16) plus the MV of HexDigit.
- The MV of HexDigits :: HexDigits NumericLiteralSeparator HexDigit is (the MV of HexDigits \times 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; 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*.

DecimalBigIntegerLiteral :: 0 BigIntLiteralSuffix

1. Return 0₂.

DecimalBigIntegerLiteral :: NonZeroDigit BigIntLiteralSuffix

1. Return the BigInt value that represents the MV of *NonZeroDigit*.

DecimalBigIntegerLiteral ::

- 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 \( Z(mv) \).

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 \no\not\not one of " or \ or LineTerminator
   \<LS>
   \<PS>
   \ EscapeSequence
   \ LineContinuation

SingleStringCharacter ::
   SourceCharacter \no\not\not one of \ or \ or LineTerminator
   \<LS>
   \<PS>
   \ EscapeSequence
   \ LineContinuation

LineContinuation ::
   \ LineTerminatorSequence

EscapeSequence ::
   CharacterEscapeSequence
   0 \lookahead \notin DecimalDigit
   HexEscapeSequence
   UnicodeEscapeSequence

A conforming implementation, when processing \textit{strict mode code}, must not extend the syntax of \textit{EscapeSequence} to include \textit{LegacyOctalEscapeSequence} or \textit{NonOctalDecimalEscapeSequence} as described in B.1.2.

CharacterEscapeSequence ::
   SingleEscapeCharacter
   NonEscapeCharacter

SingleEscapeCharacter :: \textbf{one of}
   \ \\f\n\r\t\v

NonEscapeCharacter ::
   SourceCharacter \no\not\not one of EscapeCharacter \ or \ LineTerminator

EscapeCharacter ::
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 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

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Code Unit Value</th>
<th>Unicode Character Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>0x0008</td>
<td>BACKSPACE</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>\t</td>
<td>0x0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>\n</td>
<td>0x000A</td>
<td>LINE FEED (LF)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\v</td>
<td>0x000B</td>
<td>LINE TABULATION</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>\f</td>
<td>0x000C</td>
<td>FORM FEED (FF)</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>\r</td>
<td>0x000D</td>
<td>CARRIAGE RETURN (CR)</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>0x0022</td>
<td>QUOTATION MARK</td>
<td>&quot;</td>
</tr>
<tr>
<td>’</td>
<td>0x0027</td>
<td>APOSTROPHE</td>
<td>’</td>
</tr>
<tr>
<td>\</td>
<td>0x005C</td>
<td>REVERSE SOLIDUS</td>
<td>\</td>
</tr>
</tbody>
</table>

• The 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 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]
```
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

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: /(?::)/.

12.8.5.2 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
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 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 TRV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- The TV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- The TRV of TemplateCharacter :: \ NotEscapeSequence is undefined.
- The TRV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- 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 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 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:

297
- 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).
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 ;

BreakStatement [Yield, Await] :
  break ;

ReturnStatement [Yield, Await] :
  return ;

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

```javascript
{ 1
 2 }
3
```
is also not a valid ECMAScript sentence, but is transformed by automatic semicolon insertion into the following:

```javascript
{ 1
 ;2 ;} 3;
```
which is a valid ECMAScript sentence.

The source

```javascript
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

```javascript
return
a + b
```
is transformed by automatic semicolon insertion into the following:

```javascript
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

```javascript
a = b 
++c
```
is transformed by automatic semicolon insertion into the following:

```javascript
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 `++`.
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 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 ( ).** Without a semicolon, the two lines together are treated as a CallExpression.
- **An opening square bracket [ ].** 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.
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 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

\[\text{IdentifierReference} \ldots \text{Identifier} \ldots \text{yield} \ldots \text{await} \ldots \text{Identifier} \ldots \text{yield} \ldots \text{await} \ldots \text{Identifier} \ldots \text{Name} \ldots \text{but not ReservedWord}\]
NOTE \hspace{1cm} \textbf{yield} and \textbf{await} are permitted as \textit{BindingIdentifier} in the grammar, and prohibited with \textit{static semantics} below, to prohibit automatic semicolon insertion in cases such as

\begin{verbatim}
let
await 0;
\end{verbatim}

13.1.1 Static Semantics: Early Errors

\textit{BindingIdentifier} : Identifier

- It is a Syntax Error if the code matched by this production is contained in \textit{strict mode code} and the \textit{StringValue} of \textit{Identifier} is "arguments" or "eval".

\textit{IdentifierReference} : \textit{yield}

\textit{BindingIdentifier} : \textit{yield}

\textit{LabelIdentifier} : \textit{yield}

- It is a Syntax Error if the code matched by this production is contained in \textit{strict mode code}.

\textit{IdentifierReference} : \textit{await}

\textit{BindingIdentifier} : \textit{await}

\textit{LabelIdentifier} : \textit{await}

- It is a Syntax Error if the \textit{goal symbol} of the syntactic grammar is \textit{Module}.

\textit{BindingIdentifier}[\textbf{Yield, Await}] : \textbf{yield}

- It is a Syntax Error if this production has a [Yield] Parameter.

\textit{BindingIdentifier}[\textbf{Yield, Await}] : \textbf{await}

- It is a Syntax Error if this production has an [Await] Parameter.

\textit{IdentifierReference}[\textbf{Yield, Await}] : Identifier

\textit{BindingIdentifier}[\textbf{Yield, Await}] : Identifier

\textit{LabelIdentifier}[\textbf{Yield, Await}] : Identifier

- It is a Syntax Error if this production has a [Yield] Parameter and \textit{StringValue} of \textit{Identifier} is "yield".
- It is a Syntax Error if this production has an [Await] Parameter and \textit{StringValue} of \textit{Identifier} is "await".

\textit{Identifier} : IdentifierName but not ReservedWord

- It is a Syntax Error if this phrase is contained in \textit{strict mode code} and the \textit{StringValue} of \textit{IdentifierName} is: "implements", "interface", "let", "package", "private", "protected", "public", "static", or "yield".
- It is a Syntax Error if the \textit{goal symbol} of the syntactic grammar is \textit{Module} and the \textit{StringValue} of \textit{IdentifierName} is "await".
- It is a Syntax Error if \textit{StringValue} of \textit{IdentifierName} is the same \textit{StringValue} as the \textit{StringValue} of any \textit{ReservedWord} except for \textbf{yield} or \textbf{await}.
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.

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

\[
\text{ArrayLiteral} \{\text{Yield, Await} \}:
\begin{align*}
&[\text{Elision}\_\text{opt}] \\
&[\text{ElementList}\{\text{?Yield, ?Await}\}] \\
&[\text{ElementList}\{\text{?Yield, ?Await}\}, \text{Elision}\_\text{opt}]
\end{align*}
\]

\[
\text{ElementList}\{\text{Yield, Await}\}:
\begin{align*}
&\text{Elision}\_\text{opt} \text{ AssignmentExpression} [+\text{In}, \text{?Yield, ?Await}] \\
&\text{Elision}\_\text{opt} \text{ SpreadElement}\{\text{?Yield, ?Await}\} \\
&\text{ElementList}\{\text{?Yield, ?Await}\}, \text{Elision}\_\text{opt} \text{ AssignmentExpression} [+\text{In}, \text{?Yield, ?Await}] \\
&\text{ElementList}\{\text{?Yield, ?Await}\}, \text{Elision}\_\text{opt} \text{ SpreadElement}\{\text{?Yield, ?Await}\}
\end{align*}
\]

\[
\text{AssignmentExpression} \{\text{Yield, Await}\}:
\]

\[
\text{SpreadElement}\{\text{Yield, Await}\}:
\]

13.2.5.1 Runtime Semantics: ArrayAccumulation

With parameters \textit{array} and \textit{nextIndex}.

\[
\text{Elision} :,
\]

1. Let \textit{len} be \textit{nextIndex} + 1.
2. Perform ? \text{Set}(\textit{array}, "\text{length}", F(\textit{len}), \text{true}).
3. NOTE: The above Set throws if \textit{len} exceeds \(2^{32}-1\).
4. Return \textit{len}.

\[
\text{Elision} : \text{Elision} ,
\]

1. Return the result of performing \text{ArrayAccumulation} for \textit{Elision} with arguments \textit{array} and \textit{nextIndex} + 1.

\[
\text{ElementList} : \text{Elision}\_\text{opt} \text{ AssignmentExpression}
\]

1. If \textit{Elision} is present, then
   a. Set \textit{nextIndex} to the result of performing \text{ArrayAccumulation} for \textit{Elision} with arguments \textit{array} and \textit{nextIndex}.
   b. ReturnIfAbrupt(\textit{nextIndex}).
2. Let \textit{initResult} be the result of evaluating \textit{AssignmentExpression}.
3. Let \textit{initValue} be ? \text{GetValue}(\textit{initResult}).
4. Let \textit{created} be ! \text{CreateDataPropertyOrThrow}(\textit{array}, ! \text{ToString}(F(\textit{nextIndex})), \textit{initValue}).
5. Return \textit{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]
```

```
PropertyDefinition [Yield, Await] :
   IdentifierReference[?Yield, ?Await]
   CoverInitializedName[?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. Return `IfAbrupt(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 }


AsyncMethod : `async PropertyName ( UniqueFormalParameters ) { AsyncFunctionBody }


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 \texttt{pattern} be \texttt{CodePointsToString(BodyText of RegularExpressionLiteral)}.
2. Let \texttt{flags} be \texttt{CodePointsToString(FlagText of RegularExpressionLiteral)}.
3. Return \texttt{RegExpCreate(pattern, flags)}.

### 13.2.9 Template Literals

Syntax

\[
\text{TemplateLiteral}[\text{Yield, Await, Tagged}] : \\
\text{NoSubstitutionTemplate} \\
\text{SubstitutionTemplate}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}]
\]

\[
\text{SubstitutionTemplate}[\text{Yield, Await, Tagged}] : \\
\text{TemplateHead Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}] \\
\text{TemplateSpans}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}]
\]

\[
\text{TemplateSpans}[\text{Yield, Await, Tagged}] : \\
\text{TemplateTail} \\
\text{TemplateMiddleList}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}] \\
\text{TemplateTail}
\]

\[
\text{TemplateMiddleList}[\text{Yield, Await, Tagged}] : \\
\text{TemplateMiddle Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}] \\
\text{TemplateMiddleList}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}] \\
\text{TemplateMiddle Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}]
\]

### 13.2.9.1 Static Semantics: Early Errors

\[
\text{TemplateLiteral}[\text{Yield, Await, Tagged}] : \text{NoSubstitutionTemplate}
\]

- It is a Syntax Error if the \texttt{[Tagged]} parameter was not set and \texttt{NoSubstitutionTemplate} \texttt{Contains} \texttt{NotEscapeSequence}.

\[
\text{TemplateLiteral}[\text{Yield, Await, Tagged}] : \text{SubstitutionTemplate}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}]
\]

- It is a Syntax Error if the number of elements in the result of \texttt{TemplateStrings} of \texttt{TemplateLiteral} with argument \texttt{false} is greater than \(2^{32} - 1\).

\[
\text{SubstitutionTemplate}[\text{Yield, Await, Tagged}] : \text{TemplateHead Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}]
\]

\[
\text{TemplateSpans}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}]
\]

- It is a Syntax Error if the \texttt{[Tagged]} parameter was not set and \texttt{TemplateHead} \texttt{Contains} \texttt{NotEscapeSequence}.

\[
\text{TemplateSpans}[\text{Yield, Await, Tagged}] : \text{TemplateTail}
\]

- It is a Syntax Error if the \texttt{[Tagged]} parameter was not set and \texttt{TemplateTail} \texttt{Contains} \texttt{NotEscapeSequence}.

\[
\text{TemplateMiddleList}[\text{Yield, Await, Tagged}] :
\]

\[
\text{TemplateMiddle Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}]
\]

\[
\text{TemplateMiddleList}[?\text{Yield, } ?\text{Await, } ?\text{Tagged}] \\
\text{TemplateMiddle Expression} [+\text{In, } ?\text{Yield, } ?\text{Await}]
\]

- It is a Syntax Error if the \texttt{[Tagged]} parameter was not set and \texttt{TemplateMiddle} \texttt{Contains} \texttt{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.
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(/u1D53D(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

1. Return a new empty List.

13.2.9.5 Runtime Semantics: Evaluation

1. Return the TV of NoSubstitutionTemplate as defined in 12.8.6.

NOTE 1  The string conversion semantics applied to the Expression value are like String.prototype.concat rather than the + operator.
1. Let head be the result of evaluating TemplateMiddleList.
2. ReturnIfAbrupt(head).
3. Let tail be the TV of TemplateTail as defined in 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).
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.
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
SuperProperty[Yield, ?Await]
MetaProperty
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] [ Expression[+In, ?Yield, ?Await] ]
  CallExpression[Yield, ?Await] . IdentifierName
```
Supercall \[\text{Yield, Await}\] :
  \text{super} \text{ Arguments}[?\text{Yield, ?Await}]

ImportCall \[\text{Yield, Await}\] :
  \text{import} ( \text{AssignmentExpression}[+\text{In, ?Yield, ?Await}] )

Arguments \[\text{Yield, Await}\] :
  ()
  ( \text{ArgumentList}[?\text{Yield, ?Await}] )
  ( \text{ArgumentList}[?\text{Yield, ?Await}], )

ArgumentList \[\text{Yield, Await}\] :
  \text{AssignmentExpression}[+\text{In, ?Yield, ?Await}]
  ... \text{AssignmentExpression}[+\text{In, ?Yield, ?Await}]
  \text{ArgumentList}[?\text{Yield, ?Await}], \text{AssignmentExpression}[+\text{In, ?Yield, ?Await}]
  \text{ArgumentList}[?\text{Yield, ?Await}], ... \text{AssignmentExpression}[+\text{In, ?Yield, ?Await}]

OptionalExpression \[\text{Yield, Await}\] :
  \text{MemberExpression}[?\text{Yield, ?Await}] \text{OptionalChain}[?\text{Yield, ?Await}]
  \text{CallExpression}[?\text{Yield, ?Await}] \text{OptionalChain}[?\text{Yield, ?Await}]
  \text{OptionalExpression}[?\text{Yield, ?Await}] \text{OptionalChain}[?\text{Yield, ?Await}]

OptionalChain \[\text{Yield, Await}\] :
  ?. \text{Arguments}[?\text{Yield, ?Await}]
  ?. [ \text{Expression}[+\text{In, ?Yield, ?Await}] ]
  ?. \text{IdentifierName}
  ?. \text{TemplateLiteral}[?\text{Yield, ?Await, +Tagged}]
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{Arguments}[?\text{Yield, ?Await}]
  \text{OptionalChain}[?\text{Yield, ?Await}] [ \text{Expression}[+\text{In, ?Yield, ?Await}] ]
  \text{OptionalChain}[?\text{Yield, ?Await}] . \text{IdentifierName}
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{TemplateLiteral}[?\text{Yield, ?Await, +Tagged}]

LeftHandSideExpression \[\text{Yield, Await}\] :
  \text{NewExpression}[?\text{Yield, ?Await}]
  \text{CallExpression}[?\text{Yield, ?Await}]
  \text{OptionalExpression}[?\text{Yield, ?Await}]

Supplemental Syntax

When processing an instance of the production
  CallExpression : CoverCallExpressionAndAsyncArrowHead
the interpretation of CoverCallExpressionAndAsyncArrowHead is refined using the following grammar:

CallMemberExpression \[\text{Yield, Await}\] :
  \text{MemberExpression}[?\text{Yield, ?Await}] \text{Arguments}[?\text{Yield, ?Await}]

320
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
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`.

#### 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`.

#### CallExpression : CallExpression . IdentifierName
1. Let \( \text{baseReference} \) be the result of evaluating \( \text{CallExpression} \).
2. Let \( \text{baseValue} \) be \(? \text{GetValue}(\text{baseReference})\).
3. If the code matched by this \( \text{CallExpression} \) is strict mode code, let \( \text{strict} \) be \text{true}; else let \( \text{strict} \) be \text{false}.
4. Return \(? \text{EvaluatePropertyAccessWithIdentifierKey}(\text{baseValue}, \text{IdentifierName}, \text{strict})\).

### 13.3.3 EvaluatePropertyAccessWithExpressionKey ( \( \text{baseValue} \), \( \text{expression} \), \( \text{strict} \) )

The abstract operation \( \text{EvaluatePropertyAccessWithExpressionKey} \) takes arguments \( \text{baseValue} \) (an ECMA-Script language value), \( \text{expression} \) (a Parse Node), and \( \text{strict} \) (a Boolean). It performs the following steps when called:

1. Let \( \text{propertyNameReference} \) be the result of evaluating \( \text{expression} \).
2. Let \( \text{propertyNameValue} \) be \(? \text{GetValue}(\text{propertyNameReference})\).
3. Let \( \text{bv} \) be \(? \text{RequireObjectCoercible}(\text{baseValue})\).
4. Let \( \text{propertyKey} \) be \(? \text{ToPropertyKey}(\text{propertyNameValue})\).
5. Return the Reference Record \{ [[Base]]: \( \text{bv} \), [[ReferencedName]]: \( \text{propertyKey} \), [[Strict]]: \( \text{strict} \), [[ThisValue]]: empty \}.

### 13.3.4 EvaluatePropertyAccessWithIdentifierKey ( \( \text{baseValue} \), \( \text{identifierName} \), \( \text{strict} \) )

The abstract operation \( \text{EvaluatePropertyAccessWithIdentifierKey} \) takes arguments \( \text{baseValue} \) (an ECMA-Script language value), \( \text{identifierName} \) (a Parse Node), and \( \text{strict} \) (a Boolean). It performs the following steps when called:

1. Assert: \( \text{identifierName} \) is an \( \text{IdentifierName} \).
2. Let \( \text{bv} \) be \(? \text{RequireObjectCoercible}(\text{baseValue})\).
3. Let \( \text{propertyNameString} \) be StringValue of \( \text{identifierName} \).
4. Return the Reference Record \{ [[Base]]: \( \text{bv} \), [[ReferencedName]]: \( \text{propertyNameString} \), [[Strict]]: \( \text{strict} \), [[ThisValue]]: empty \}.

### 13.3.5 The new Operator

#### 13.3.5.1 Runtime Semantics: Evaluation

**NewExpression:** \( \text{new NewExpression} \)

1. Return \(? \text{EvaluateNew}(\text{NewExpression}, \text{empty})\).

**MemberExpression:** \( \text{new MemberExpression Arguments} \)

1. Return \(? \text{EvaluateNew}(\text{MemberExpression}, \text{Arguments})\).

#### 13.3.5.1.1 EvaluateNew ( \( \text{constructExpr} \), \( \text{arguments} \) )

The abstract operation \( \text{EvaluateNew} \) takes arguments \( \text{constructExpr} \) and \( \text{arguments} \). It performs the following steps when called:

1. Assert: \( \text{constructExpr} \) is either a \( \text{NewExpression} \) or a \( \text{MemberExpression} \).
2. Assert: \( \text{arguments} \) is either \text{empty} or an \( \text{Arguments} \).
3. Let \( \text{ref} \) be the result of evaluating \( \text{constructExpr} \).
4. Let \( \text{constructor} \) be \(? \text{GetValue}(\text{ref})\).
5. If \( \text{arguments} \) is \text{empty}, let \( \text{argList} \) be a new empty \( \text{List} \).
6. Else,
   a. Let \text{argList} be ? \text{ArgumentListEvaluation} of arguments.
7. If \text{IsConstructor(constructor)} is false, throw a TypeError exception.

### 13.3.6 Function Calls

#### 13.3.6.1 Runtime Semantics: Evaluation

**CallExpression : CoverCallExpressionAndAsyncArrowHead**

1. Let expr be \text{CoveredCallExpression} of \text{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, \text{IsPropertyReference(ref)} is false, and ref.[[ReferencedName]] is "eval", then
   a. If SameValue(func, %eval%) is true, then
      i. Let argList be ? \text{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 ? \text{PerformEval}(evalArg, evalRealm, strictCaller, true).
7. Let thisCall be this CallExpression.
8. Let tailCall be IsInTailPosition(thisCall).

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 \text{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.
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().
   5. If IsConstructor(func) is false, throw a TypeError exception.
   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 \( \text{envRec} \) be \( \text{GetThisEnvironment()} \).
2. Assert: \( \text{envRec} \) is a function Environment Record.
3. Let \( \text{activeFunction} \) be \( \text{envRec}.[[\text{FunctionObject}]] \).
4. Assert: \( \text{activeFunction} \) is an ECMAScript function object.
5. Let \( \text{superConstructor} \) be ! \( \text{activeFunction}.[[\text{GetPrototypeOf}]]() \).
6. Return \( \text{superConstructor} \).

13.3.7.3 MakeSuperPropertyReference ( \( \text{actualThis}, \text{propertyKey}, \text{strict} \) )

The abstract operation MakeSuperPropertyReference takes arguments \( \text{actualThis}, \text{propertyKey}, \) and \( \text{strict} \). It performs the following steps when called:

1. Let \( \text{env} \) be \( \text{GetThisEnvironment()} \).
2. Assert: \( \text{env}.\text{HasSuperBinding()} \) is \( \text{true} \).
3. Let \( \text{baseValue} \) be ? \( \text{env}.\text{GetSuperBase()} \).
4. Let \( \text{bv} \) be ? \( \text{RequireObjectCoercible}(\text{baseValue}) \).
5. Return the Reference Record \{ [[Base]]: \( \text{bv} \), [[ReferencedName]]: \( \text{propertyKey} \), [[Strict]]: \( \text{strict} \), [[ThisValue]]: \( \text{actualThis} \) \}.
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 \( \text{ref} \) be the result of evaluating AssignmentExpression.
2. Let \( \text{arg} \) be ? \( \text{GetValue}(\text{ref}) \).
3. Return a List whose sole element is \( \text{arg} \).

ArgumentList : ... AssignmentExpression

1. Let \( \text{list} \) be a new empty List.
2. Let \( \text{spreadRef} \) be the result of evaluating AssignmentExpression.
3. Let \( \text{spreadObj} \) be ? \( \text{GetValue}(\text{spreadRef}) \).
4. Let \( \text{iteratorRecord} \) be ? \( \text{GetIterator}(\text{spreadObj}) \).
5. Repeat,
   a. Let \( \text{next} \) be ? \( \text{IteratorStep}(\text{iteratorRecord}) \).
   b. If \( \text{next} \) is false, return \( \text{list} \).
   c. Let \( \text{nextArg} \) be ? \( \text{IteratorValue}(\text{next}) \).
   d. Append \( \text{nextArg} \) as the last element of \( \text{list} \).

ArgumentList : ArgumentList , AssignmentExpression
1. Let \texttt{precedingArgs} be \texttt{ArgumentListEvaluation} of \texttt{ArgumentList}.
2. Let \texttt{ref} be the result of evaluating \texttt{AssignmentExpression}.
3. Let \texttt{arg} be \texttt{GetValue(ref)}.
4. Append \texttt{arg} to the end of \texttt{precedingArgs}.
5. Return \texttt{precedingArgs}.

\texttt{ArgumentList : ArgumentList , ... AssignmentExpression}

1. Let \texttt{precedingArgs} be \texttt{ArgumentListEvaluation} of \texttt{ArgumentList}.
2. Let \texttt{spreadRef} be the result of evaluating \texttt{AssignmentExpression}.
3. Let \texttt{iteratorRecord} be \texttt{GetIterator(? GetValue(spreadRef))}.
4. Repeat,
   a. Let \texttt{next} be \texttt{IteratorStep(iteratorRecord)}.
   b. If \texttt{next} is \texttt{false}, return \texttt{precedingArgs}.
   c. Let \texttt{nextArg} be \texttt{IteratorValue(next)}.
   d. Append \texttt{nextArg} as the last element of \texttt{precedingArgs}.

\texttt{TemplateLiteral : NoSubstitutionTemplate}

1. Let \texttt{templateLiteral} be this \texttt{TemplateLiteral}.
2. Let \texttt{siteObj} be \texttt{GetTemplateObject(templateLiteral)}.
3. Return a \texttt{List} whose sole element is \texttt{siteObj}.

\texttt{TemplateLiteral : SubstitutionTemplate}

1. Let \texttt{templateLiteral} be this \texttt{TemplateLiteral}.
2. Let \texttt{siteObj} be \texttt{GetTemplateObject(templateLiteral)}.
3. Let \texttt{remaining} be \texttt{ArgumentListEvaluation} of \texttt{SubstitutionTemplate}.
4. Return a \texttt{List} whose first element is \texttt{siteObj} and whose subsequent elements are the elements of \texttt{remaining}.

\texttt{SubstitutionTemplate : TemplateHead Expression TemplateSpans}

1. Let \texttt{firstSubRef} be the result of evaluating \texttt{Expression}.
2. Let \texttt{firstSub} be \texttt{GetValue(firstSubRef)}.
3. Let \texttt{restSub} be \texttt{SubstitutionEvaluation of TemplateSpans}.
4. Assert: \texttt{restSub} is a \texttt{List}.
5. Return a \texttt{List} whose first element is \texttt{firstSub} and whose subsequent elements are the elements of \texttt{restSub}. \texttt{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 \texttt{?. .}.

### 13.3.9.1 Runtime Semantics: Evaluation

\texttt{OptionalExpression :}

\texttt{MemberExpression OptionalChain}

1. Let \texttt{baseReference} be the result of evaluating \texttt{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)`.

**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`.
3. Let `newValue` be ? `GetValue(newReference)`.
4. Let `thisChain` be this `OptionalChain`.
5. Let `tailCall` be `IsInTailPosition(thisChain)`.
OptionalChain : OptionalChain  [  Expression  ]

1. Let \texttt{optionalChain} be \texttt{OptionalChain}.
2. Let \texttt{newReference} be \texttt{? ChainEvaluation} of \texttt{optionalChain} with arguments \texttt{baseValue} and \texttt{baseReference}.
3. Let \texttt{newValue} be \texttt{? GetValue(newReference)}.
4. If the code matched by this \texttt{OptionalChain} is strict mode code, let \texttt{strict} be \texttt{true}; else let \texttt{strict} be \texttt{false}.
5. Return \texttt{? EvaluatePropertyAccessWithExpressionKey(newValue, Expression, strict)}.

OptionalChain : OptionalChain . IdentifierName

1. Let \texttt{optionalChain} be \texttt{OptionalChain}.
2. Let \texttt{newReference} be \texttt{? ChainEvaluation} of \texttt{optionalChain} with arguments \texttt{baseValue} and \texttt{baseReference}.
3. Let \texttt{newValue} be \texttt{? GetValue(newReference)}.
4. If the code matched by this \texttt{OptionalChain} is strict mode code, let \texttt{strict} be \texttt{true}; else let \texttt{strict} be \texttt{false}.
5. Return \texttt{? EvaluatePropertyAccessWithIdentifierKey(newValue, IdentifierName, strict)}.

### 13.3.10 Import Calls

#### 13.3.10.1 Runtime Semantics: Evaluation

ImportCall : import ( AssignmentExpression )

1. Let \texttt{referencingScriptOrModule} be \texttt{! GetActiveScriptOrModule()}.  
2. Let \texttt{argRef} be the result of evaluating \texttt{AssignmentExpression}.
3. Let \texttt{specifier} be \texttt{? GetValue(argRef)}.
4. Let \texttt{promiseCapability} be \texttt{! NewPromiseCapability(%Promise%)}.  
5. Let \texttt{specifierString} be \texttt{ToString(specifier)}.
6. IfAbruptRejectPromise(specifierString, promiseCapability).
7. Perform \texttt{! HostImportModuleDynamically(referencingScriptOrModule, specifierString, promiseCapability)}.
8. Return \texttt{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 \texttt{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 \texttt{TemplateLiteral}.

#### 13.3.11.1 Runtime Semantics: Evaluation

MemberExpression : MemberExpression TemplateLiteral

1. Let \texttt{tagRef} be the result of evaluating \texttt{MemberExpression}.
2. Let \texttt{tagFunc} be \texttt{? GetValue(tagRef)}.
3. Let \texttt{thisCall} be this \texttt{MemberExpression}.
4. Let \texttt{tailCall} be \texttt{IsInTailPosition(thisCall)}.
5. Return \texttt{? EvaluateCall(tagFunc, tagRef, TemplateLiteral, tailCall)}.

CallExpression : CallExpression  TemplateLiteral

1. Let \texttt{tagRef} be the result of evaluating \texttt{CallExpression}.

---

329
2. Let \( tagFunc \) be \( \text{GetValue}(tagRef) \).
3. Let \( thisCall \) be this \( \text{CallExpression} \).
4. Let \( tailCall \) be \( \text{IsInTailPosition}(thisCall) \).
5. Return \( \text{? EvaluateCall}(tagFunc, tagRef, \text{TemplateLiteral}, tailCall) \).

13.3.12 Meta Properties

13.3.12.1 Runtime Semantics: Evaluation

NewTarget : new . target

1. Return \( \text{GetNewTarget()} \).

ImportMeta : import . meta

1. Let \( module \) be \( ! \text{GetActiveScriptOrModule}() \).
2. Assert: \( module \) is a Source Text Module Record.
3. Let \( importMeta \) be \( module.[[\text{ImportMeta}]] \).
4. If \( importMeta \) is empty, then
   a. Set \( importMeta \) to \( ! \text{OrdinaryObjectCreate}(\text{null}) \).
   b. Let \( importMetaValues \) be \( ! \text{HostGetImportMetaProperties}(module) \).
   c. For each \( \text{Record} \{ [[\text{Key}]], [[\text{Value}]] \} \ p \) of \( importMetaValues \), do
      i. Perform \( ! \text{CreateDataPropertyOrThrow}(importMeta, p.[[\text{Key}]], p.[[\text{Value}]]) \).
   d. Perform \( ! \text{HostFinalizeImportMeta}(importMeta, module) \).
   e. Set \( module.[[\text{ImportMeta}]] \) 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., \( \text{IsPropertyKey} \) 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**

\[
\text{UpdateExpression} \quad [\text{Yield, Await}] \\
\quad : \\
\quad \text{LeftHandSideExpression} \quad [?\text{Yield, ?Await}] \\
\quad \text{LeftHandSideExpression} \quad [?\text{Yield, ?Await}] \quad \text{[no LineTerminator here]} \quad ++ \\
\quad \text{LeftHandSideExpression} \quad [?\text{Yield, ?Await}] \quad \text{[no LineTerminator here]} \quad -- \\
\quad \quad ++ \text{UnaryExpression} [?\text{Yield, ?Await}] \\
\quad \quad -- \text{UnaryExpression} [?\text{Yield, ?Await}]
\]

#### 13.4.1 Static Semantics: Early Errors

**UpdateExpression** :

- \[\text{LeftHandSideExpression} \quad \text{++}
- \text{LeftHandSideExpression} \quad \text{--}

- It is an early Syntax Error if \text{AssignmentTargetType} of \text{LeftHandSideExpression} is not simple.

**UpdateExpression** :

- \[\text{++ \text{UnaryExpression}}
- \text{-- \text{UnaryExpression}}

- It is an early Syntax Error if \text{AssignmentTargetType} of \text{UnaryExpression} is not simple.

#### 13.4.2 Postfix Increment Operator

**13.4.2.1 Runtime Semantics: Evaluation**

**UpdateExpression** : \[\text{LeftHandSideExpression} \quad \text{++}

1. Let \(lhs\) be the result of evaluating \text{LeftHandSideExpression}.
2. Let \(oldValue\) be \(\text{?ToNumeric(?GetValue(lhs))}\).
3. Let \(newValue\) be \(\text{!Type(oldValue)::add(oldValue, Type(oldValue)::unit)}\).
4. Perform \(\text{?PutValue(lhs, newValue)}\).
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 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

\textbf{UnaryExpression} : \textbf{delete} \textbf{UnaryExpression}

- It is a Syntax Error if the \textit{UnaryExpression} is contained in \textbf{strict mode code} and the derived \textit{UnaryExpression} is \textbf{PrimaryExpression} : \textbf{IdentifierReference}.

- It is a Syntax Error if the derived \textit{UnaryExpression} is \textbf{PrimaryExpression} : \textbf{CoverParenthesizedExpressionAndArrowParameterList} and \textbf{CoverParenthesizedExpressionAndArrowParameterList} ultimately derives a phrase that, if used in place of \textit{UnaryExpression}, would produce a Syntax Error according to these rules. This rule is recursively applied.

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

13.5.1.2 Runtime Semantics: Evaluation

\textbf{UnaryExpression} : \textbf{delete} \textbf{UnaryExpression}

1. Let \textit{ref} be the result of evaluating \textit{UnaryExpression}.
2. \textbf{ReturnIfAbrupt}(\textit{ref}).
3. If \textit{ref} is not a Reference Record, return \textbf{true}.
4. If \textbf{IsUnresolvableReference}(\textit{ref}) is \textbf{true}, then
   a. \texttt{Assert: ref.\texttt{[[Strict]]} is false}.
   b. Return \textbf{true}.
5. If \textbf{IsPropertyReference}(\textit{ref}) is \textbf{true}, then
   a. If \textbf{IsSuperReference}(\textit{ref}) is \textbf{true}, throw a \texttt{ReferenceError} exception.
   b. Let \textit{baseObj} be ! \texttt{ToObject}(\textit{ref.\texttt{[[Base]]}}).
   c. Let \textit{deleteStatus} be ? \textit{baseObj.\texttt{[[Delete]]}}(\textit{ref.\texttt{[[ReferencedName]]}}).
   d. If \textit{deleteStatus} is \textbf{false} and \textit{ref.\texttt{[[Strict]]}} is \textbf{true}, throw a \texttt{TypeError} exception.
   e. Return \textit{deleteStatus}.
6. Else,
   a. Let \textit{base} be \textit{ref.\texttt{[[Base]]}}.
   b. \texttt{Assert: base} is an Environment Record.
   c. Return ? \textit{base.DeleteBinding}(\textit{ref.\texttt{[[ReferencedName]]}}).

\textbf{NOTE 1} When a \texttt{delete} operator occurs within \textbf{strict mode code}, a \texttt{SyntaxError} exception is thrown if its \textit{UnaryExpression} is a direct reference to a variable, function argument, or function name. In addition, if a \texttt{delete} operator occurs within \textbf{strict mode code} and the property to be deleted has the attribute \{ \texttt{[[Configurable]]} : \texttt{false} \} (or otherwise cannot be deleted), a \texttt{TypeError} exception is thrown.

\textbf{NOTE 2} The object that may be created in step 5.b is not accessible outside of the above abstract operation and the \texttt{ordinary object} \texttt{[[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>
<thead>
<tr>
<th>Type of <code>val</code></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>Symbol</td>
<td>&quot;symbol&quot;</td>
</tr>
<tr>
<td>BigInt</td>
<td>&quot;bigint&quot;</td>
</tr>
<tr>
<td>Object (does not implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (implements [[Call]])</td>
<td>&quot;function&quot;</td>
</tr>
</tbody>
</table>

**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

NOTE The unary - operator converts its operand to Number type and then negates it. Negating +0\(_F\) produces -0\(_F\), and negating -0\(_F\) produces +0\(_F\).

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


13.7 Multiplicative Operators

Syntax

MultiplicativeExpression[Yield, Await] :
  ExponentiationExpression[Yield, Await]

MultiplicaitveOperator : 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 : MultiplicaitveExpression MultiplicaitveOperator MultiplicaitveOperator ExponentiationExpression

1. Let opText be the source text matched by MultiplicaitveOperator.

13.8 Additive Operators

Syntax

AdditiveExpression[Yield, Await] :
  MultiplicaitveExpression[Yield, Await]
  AdditiveExpression[Yield, Await] + MultiplicaitveExpression[Yield, Await]
  AdditiveExpression[Yield, Await] - MultiplicaitveExpression[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

\[
\text{AdditiveExpression : AdditiveExpression} + \text{MultiplicativeExpression}
\]

1. Return \(?\text{EvaluateStringOrNumericBinaryExpression(AdditiveExpression, +, MultiplicativeExpression)}.\)

13.8.2 The Subtraction Operator (−)

NOTE The − operator performs subtraction, producing the difference of its operands.

13.8.2.1 Runtime Semantics: Evaluation

\[
\text{AdditiveExpression : AdditiveExpression} - \text{MultiplicativeExpression}
\]

1. Return \(?\text{EvaluateStringOrNumericBinaryExpression(AdditiveExpression, -, MultiplicativeExpression)}.\)

13.9 Bitwise Shift Operators

Syntax

\[
\text{ShiftExpression}[\text{Yield, Await}] : \\
\quad \text{AdditiveExpression}[\text{Yield, Await}] \\
\quad \text{ShiftExpression}[\text{Yield, Await}] \ll \text{AdditiveExpression}[\text{Yield, Await}] \\
\quad \text{ShiftExpression}[\text{Yield, Await}] \gg \text{AdditiveExpression}[\text{Yield, Await}] \\
\quad \text{ShiftExpression}[\text{Yield, Await}] \gg\gg \text{AdditiveExpression}[\text{Yield, Await}]
\]

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

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

1. Return \(?\text{EvaluateStringOrNumericBinaryExpression(ShiftExpression, <<, AdditiveExpression)}.\)

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


13.9 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


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]

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 \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{ShiftExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
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 \( \text{RelationalExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{ShiftExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
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 \( \text{RelationalExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{ShiftExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
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 \( \text{RelationalExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{ShiftExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
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 \( \text{RelationalExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{ShiftExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
5. Return ? `InstanceofOperator(lval, rval)`.

**RelationalExpression : RelationalExpression in ShiftExpression**

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

### 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 \(\text{Type}(\ target)\) is not Object, throw a \(\text{TypeError}\) exception.
2. Let \(\text{instOfHandler}\) be \(?\ \text{GetMethod}(\ target, \ @@\text{hasInstance})\).
3. If \(\text{instOfHandler}\) is not undefined, then
   a. Return \(!\ \text{ToBoolean}(\ ?\ \text{Call}(\text{instOfHandler}, \ \ target, \ «\ V »))\).
4. If \(\text{IsCallable}(\ target)\) is false, throw a \(\text{TypeError}\) exception.
5. Return \(?\ \text{OrdinaryHasInstance}(\ target, \ V)\).

**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**

\[\text{EqualityExpression} [\text{In}, \ \text{Yield}, \ \text{Await}] : \text{RelationalExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}]\]

\[\text{EqualityExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}] = \text{RelationalExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}]\]

\[\text{EqualityExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}] \neq \text{RelationalExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}]\]

\[\text{EqualityExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}] \text{=== RelationalExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}]\]

\[\text{EqualityExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}] \text{!== RelationalExpression} [?\text{In}, \ ?\text{Yield}, \ ?\text{Await}]\]

### 13.11.1 Runtime Semantics: Evaluation

**EqualityExpression : EqualityExpression == RelationalExpression**

1. Let \(lref\) be the result of evaluating \(\text{EqualityExpression}\).
2. Let \(lval\) be \(?\ \text{GetValue}(lref)\).
3. Let \(rref\) be the result of evaluating \(\text{RelationalExpression}\).
4. Let \(rval\) be \(?\ \text{GetValue}(rref)\).
5. Return the result of performing Abstract Equality Comparison \(\text{real} == \text{loal}\).

**EqualityExpression : EqualityExpression \neq RelationalExpression**
1. Let \( lref \) be the result of evaluating \( \text{EqualityExpression} \).
2. Let \( lval \) be \( ? \ \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{RelationalExpression} \).
4. Let \( rval \) be \( ? \ \text{GetValue}(rref) \).
5. Let \( r \) be the result of performing \( \text{Abstract Equality Comparison} \) \( rval \ == \ lval \).
6. \( \text{ReturnIfAbrupt}(r) \).
7. If \( r \) is \text{true}, return \text{false}. Otherwise, return \text{true}.

\text{EqualityExpression} : \text{EqualityExpression} \ == \ \text{RelationalExpression} \\

1. Let \( lref \) be the result of evaluating \( \text{EqualityExpression} \).
2. Let \( lval \) be \( ? \ \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{RelationalExpression} \).
4. Let \( rval \) be \( ? \ \text{GetValue}(rref) \).
5. Return the result of performing \( \text{Strict Equality Comparison} \) \( rval \ == \ lval \).

\text{EqualityExpression} : \text{EqualityExpression} \ !== \ \text{RelationalExpression} \\

1. Let \( lref \) be the result of evaluating \( \text{EqualityExpression} \).
2. Let \( lval \) be \( ? \ \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{RelationalExpression} \).
4. Let \( rval \) be \( ? \ \text{GetValue}(rref) \).
5. Let \( r \) be the result of performing \( \text{Strict Equality Comparison} \) \( rval \ == \ lval \).
6. \text{Assert: } r \text{ is a normal completion.}
7. If \( r.[[\text{Value}]] \) is \text{true}, return \text{false}. Otherwise, return \text{true}.

\text{NOTE 1} \quad \text{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: !\( a \) == !\( b \).

\text{NOTE 2} \quad \text{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 \( B \).

\text{NOTE 3} \quad \text{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:}

- \( \text{new String("a")} == "a" \) and \( "a" == \text{new String("a")} \) are both \text{true}.
- \( \text{new String("a")} == \text{new String("a")} \) is \text{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

\[
\text{BitwiseANDExpression} [\text{In, Yield, Await}] :
\]

\[
\text{EqualityExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{BitwiseANDExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}] \& \text{ EqualityExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{BitwiseXORExpression} [\text{In, Yield, Await}] :
\]

\[
\text{BitwiseANDExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{BitwiseXORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}] \^ \text{ BitwiseANDExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{BitwiseORExpression} [\text{In, Yield, Await}] :
\]

\[
\text{BitwiseXORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{BitwiseORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}] | \text{ BitwiseXORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

13.12.1 Runtime Semantics: Evaluation

\text{BitwiseANDExpression} : \text{ BitwiseANDExpression} \& \text{ EqualityExpression}

1. Return ? \text{EvaluateStringOrNumericBinaryExpression}(\text{BitwiseANDExpression, \&, EqualityExpression}).

\text{BitwiseXORExpression} : \text{BitwiseXORExpression} ^ \text{ BitwiseANDExpression}

1. Return ? \text{EvaluateStringOrNumericBinaryExpression}(\text{BitwiseXORExpression, ^, BitwiseANDExpression}).

\text{BitwiseORExpression} : \text{BitwiseORExpression} | \text{ BitwiseXORExpression}

1. Return ? \text{EvaluateStringOrNumericBinaryExpression}(\text{BitwiseORExpression, |, BitwiseXORExpression}).

13.13 Binary Logical Operators

Syntax

\[
\text{LogicalANDExpression} [\text{In, Yield, Await}] :
\]

\[
\text{BitwiseORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{LogicalANDExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}] \&\& \text{ BitwiseORExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\[
\text{LogicalORExpression} [\text{In, Yield, Await}] :
\]

\[
\text{LogicalANDExpression} [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]
LogicalORExpression [?In, ?Yield, ?Await] || LogicalANDEXpression [?In, ?Yield, ?Await]

CoalesceExpression [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.

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.

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.

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.


**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

**AssignmentExpression** : `In`, `Yield`, `Await`  

- `ConditionalExpression` : `ShortCircuitExpression` ? `AssignmentExpression` : `AssignmentExpression`
- `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`, `**=`

**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 lref be the result of evaluating AssignmentExpression.
4. Let lval be ? GetValue(lref).
5. Let rval be ? GetValue(rval).
7. 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:
7. Let \( r \) be \( \text{ApplyStringOrNumericBinaryOperator}(\text{lval}, \text{opText}, \text{rval}) \).
8. Perform \( \text{? PutValue} (\text{lref}, r) \).
9. Return \( r \).

**AssignmentExpression** : **LeftHandSideExpression** &amp;= **AssignmentExpression**

1. Let \( \text{lref} \) be the result of evaluating \( \text{LeftHandSideExpression} \).
2. Let \( \text{lval} \) be \( \text{? GetValue} (\text{lref}) \).
3. Let \( \text{lbool} \) be \( \text{! ToBoolean} (\text{lval}) \).
4. If \( \text{lbool} \) is \( \text{false} \), return \( \text{lval} \).
5. If \( \text{IsAnonymousFunctionDefinition} (\text{AssignmentExpression}) \) is \( \text{true} \) and \( \text{IsIdentifierRef} \) of \( \text{LeftHandSideExpression} \) is \( \text{true} \), then
   a. Let \( \text{rval} \) be \( \text{NamedEvaluation} \) of \( \text{AssignmentExpression} \) with argument \( \text{lref}[[\text{ReferencedName}]] \).
6. Else,
   a. Let \( \text{rref} \) be the result of evaluating \( \text{AssignmentExpression} \).
   b. Let \( \text{rval} \) be \( \text{? GetValue} (\text{rref}) \).
7. Perform \( \text{? PutValue} (\text{lref}, \text{rval}) \).
8. Return \( \text{rval} \).

**AssignmentExpression** : **LeftHandSideExpression** ||= **AssignmentExpression**

1. Let \( \text{lref} \) be the result of evaluating \( \text{LeftHandSideExpression} \).
2. Let \( \text{lval} \) be \( \text{? GetValue} (\text{lref}) \).
3. Let \( \text{lbool} \) be \( \text{! ToBoolean} (\text{lval}) \).
4. If \( \text{lbool} \) is \( \text{true} \), return \( \text{lval} \).
5. If \( \text{IsAnonymousFunctionDefinition} (\text{AssignmentExpression}) \) is \( \text{true} \) and \( \text{IsIdentifierRef} \) of \( \text{LeftHandSideExpression} \) is \( \text{true} \), then
   a. Let \( \text{rval} \) be \( \text{NamedEvaluation} \) of \( \text{AssignmentExpression} \) with argument \( \text{lref}[[\text{ReferencedName}]] \).
6. Else,
   a. Let \( \text{rref} \) be the result of evaluating \( \text{AssignmentExpression} \).
   b. Let \( \text{rval} \) be \( \text{? GetValue} (\text{rref}) \).
7. Perform \( \text{? PutValue} (\text{lref}, \text{rval}) \).
8. Return \( \text{rval} \).
AssignmentExpression : LeftHandSideExpression ??= AssignmentExpression

1. Let \( lref \) be the result of evaluating \( \text{LeftHandSideExpression} \).
2. Let \( lval \) be ? GetValue(\( lref \)).
3. If \( lval \) is neither \( \text{undefined} \) nor \( \text{null} \), return \( lval \).
4. If \( \text{IsAnonymousFunctionDefinition}(\text{AssignmentExpression}) \) is \( \text{true} \) and \( \text{IsIdentifierRef} \) of \( \text{LeftHandSideExpression} \) is \( \text{true} \), then
   a. Let \( rval \) be \( \text{NamedEvaluation} \) of \( \text{AssignmentExpression} \) with argument \( lref.\\text{[[ReferencedName]]} \).
5. Else,
   a. Let \( rref \) be the result of evaluating \( \text{AssignmentExpression} \).
      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 is an unresolvable reference. If it is, a \( \text{ReferenceError} \) exception is thrown. Additionally, it is a runtime error if the \( lref \) in step 8, 7, 6 is a reference to a \( \text{data property} \) with the attribute value \{ \[\text{Writable}\]: \( \text{false} \}, to an \( \text{accessor property} \) with the attribute value \{ \[\text{Set}\]: \( \text{undefined} \}, or to a non-existent property of an object for which the \( \text{IsExtensible} \) predicate returns the value \( \text{false} \). In these cases a \( \text{TypeError} \) exception is thrown.

13.15.3 ApplyStringOrNumericBinaryOperator ( \( lval, opText, rval \) )

The abstract operation \( \text{ApplyStringOrNumericBinaryOperator} \) 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 ? \( \text{ToPrimitive}(lval) \).
   b. Let \( rprim \) be ? \( \text{ToPrimitive}(rval) \).
   c. If \( \text{Type}(lprim) \) is String or \( \text{Type}(rprim) \) is String, then
      i. Let \( lstr \) be ? \( \text{ToString}(lprim) \).
      ii. Let \( rstr \) be ? \( \text{ToString}(rprim) \).
      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 ? \( \text{ToNumeric}(lval) \).
5. Let \( rnum \) be ? \( \text{ToNumeric}(rval) \).
6. If \( \text{Type}(lnum) \) is different from \( \text{Type}(rnum) \), throw a \( \text{TypeError} \) exception.
7. Let \( T \) be \( \text{Type}(lnum) \).
8. Let \( operation \) be the abstract operation associated with \( opText \) in the following table:

\[
\begin{array}{ll}
\text{opText} & \text{operation} \\
** & T::exponentiate \\
* & T::multiply \\
/ & T::divide \\
\end{array}
\]
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] :
  { }
```
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.
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** : `{ }`

2. Return NormalCompletion(`empty`).

**ObjectAssignmentPattern** :

```
{ AssignmentPropertyList }
{ AssignmentPropertyList , }
```

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`.350
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 }

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 }

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 $propertyNames$ 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 $propertyNames$.
4. Return $propertyNames$.

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($defaultValue$).
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%).
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.
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.


**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

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 \( ? \text{GetV}(\text{value}, \text{propertyName}) \).
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 \( ? \text{NamedEvaluation of} \text{Initializer} \text{ with argument} lref.\{[\text{ReferencedName}]\} \).
   b. Else,
      i. Let `defaultValue` be the result of evaluating `Initializer`.
      ii. Let `rhsValue` be \( ? \text{GetValue}(\text{defaultValue}) \).
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 \( ? \text{PutValue}(\text{lref}, \text{rhsValue}) \).

### 13.16 Comma Operator ( , )

**Syntax**

\[
\text{Expression} \{[\text{In, Yield, Await}]\} : \\
\text{AssignmentExpression} \{[?\text{In}, ?\text{Yield, ?Await}]\} \\
\text{Expression} \{[?\text{In, ?Yield, ?Await}]\} , \text{AssignmentExpression} \{[?\text{In, ?Yield, ?Await}]\}
\]

### 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**

\[
\text{Statement} \{[\text{Yield, Await, Return}]\} : \\
\text{BlockStatement} \{[?\text{Yield, ?Await, ?Return}]\} \\
\text{VariableStatement} \{[?\text{Yield, ?Await}]\}
\]
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 :
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**

```plaintext
BlockStatement [Yield, Await, Return] :
  Block [?Yield, ?Await, ?Return]

Block [Yield, Await, Return] :

StatementList [Yield, Await, Return] :
  StatementListItem [?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:

```javascript
    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]

LexicalBinding [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]
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).

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] } 

ArrayBindingPattern [Yield, Await] :
  [ Elision opt BindingRestElement [?Yield, ?Await] opt ]

BindingRestProperty [Yield, Await] :
  ... BindingIdentifier [Yield, ?Await]

BindingPropertyList [Yield, Await] :
  BindingProperty [?Yield, ?Await]

BindingElementList [Yield, Await] :
  BindingElisionElement [?Yield, ?Await]

BindingElisionElement [Yield, Await] :
  Elision opt BindingElement [?Yield, ?Await]

BindingProperty [Yield, Await] :
  SingleNameBinding [?Yield, ?Await]

BindingElement [Yield, Await] :
  SingleNameBinding [?Yield, ?Await]

SingleNameBinding [Yield, Await] :

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%).
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. IfInitializer 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 `GetValue(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] :
  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 if's

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 (complementation, labelSet)**

The abstract operation LoopContinues takes arguments complementation and labelSet. It performs the following steps when called:

1. If complementation.[[Type]] is normal, return true.
2. If complementation.[[Type]] is not continue, return false.
3. If complementation.[[Target]] is empty, return true.
4. If complementation.[[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**

\[\text{DoWhileStatement} [\text{Yield, Await, Return}] : \]

#### 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**

\[\text{WhileStatement} [\text{Yield, Await, Return}] : \]
\[\text{while ( Expression [+In, ?Yield, ?Await] ) Statement [?Yield, ?Await, ?Return]} \]

367
WhileStatement: \(\text{while } (\text{Expression}) \text{ Statement}\)

- It is a Syntax Error if \(\text{IsLabelledFunction(Statement)}\) 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 \(\text{labelSet}\).

**WhileStatement**: \(\text{while } (\text{Expression}) \text{ Statement}\)

1. Let \(V\) be undefined.
2. Repeat,
   a. Let \(\text{exprRef}\) be the result of evaluating \(\text{Expression}\).
   b. Let \(\text{exprValue}\) be \(\text{GetValue(\text{exprRef})}\).
   c. If \(\text{! ToBoolean(\text{exprValue})}\) is false, return NormalCompletion(\(V\)).
   d. Let \(\text{stmtResult}\) be the result of evaluating \(\text{Statement}\).
   e. If \(\text{LoopContinues(\text{stmtResult}, \text{labelSet})}\) is false, return Completion(UpdateEmpty(\(\text{stmtResult}, V\))).
   f. If \(\text{stmtResult}.[\text{Value}]\) is not empty, set \(V\) to \(\text{stmtResult}.[\text{Value}]\).

### 14.7.4 The for Statement

**Syntax**

\[
\text{ForStatement}[\text{Yield, Await, Return}] : \\
\text{for } [\text{lookahead \#let [\emptyset]} \text{ Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Statement}[\text{+In, ?Yield, ?Await, ?Return}] \\
\text{for } [\text{var VariableDeclarationList}[\text{~In, ?Yield, ?Await} \text{; Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Statement}[\text{+In, ?Yield, ?Await, ?Return}] \\
\text{for } [\text{LexicalDeclaration}[\text{~In, ?Yield, ?Await} \text{ Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Expression}[\text{+In, ?Yield, ?Await} \text{ opt }\text{; Statement}[\text{+In, ?Yield, ?Await, ?Return}] \\
\]

**14.7.4.1 Static Semantics: Early Errors**

**ForStatement**: 

\[
\text{for } [\text{Expression opt }; \text{Expression opt }; \text{Expression opt }\text{; Statement} \\
\text{for } [\text{var VariableDeclarationList}; \text{Expression opt }; \text{Expression opt }\text{; Statement} \\
\text{for } [\text{LexicalDeclaration Expression opt}; \text{Expression opt }\text{; Statement} \\
\]

- It is a Syntax Error if \(\text{IsLabelledFunction(Statement)}\) is true.

**NOTE** It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

**ForStatement**: \(\text{for } [\text{LexicalDeclaration Expression opt}; \text{Expression opt }\text{; Statement}\)
• 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 \textit{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 \textit{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 \textit{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

\[ ForInOfStatement \] \( [\text{Yield}, \text{Await}, \text{Return}] \) :
\[ \text{for} \ ( [\text{lookahead} = \text{let}] \ [\text{LeftHandSideExpression} \ [\text{?Yield}, \text{?Await}] \ \text{in} \ \text{Expression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \) 
\[ \text{for} \ ( \text{var} \ \text{ForBinding} \ [\text{?Yield}, \text{?Await}] \ \text{in} \ \text{Expression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \) 
\[ \text{for} \ ( \text{ForDeclaration} \ [\text{?Yield}, \text{?Await}] \ \text{in} \ \text{Expression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \) 
\[ \text{for} \ ( [\text{lookahead} \notin \{\text{let, async of}\}] \ \text{LeftHandSideExpression} \ [\text{?Yield}, \text{?Await}] \ \text{of} \ \text{AssignmentExpression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \) 
\[ \text{for} \ ( \text{var} \ \text{ForBinding} \ [\text{?Yield}, \text{?Await}] \ \text{of} \ \text{AssignmentExpression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \) 
\[ \text{for} \ ( \text{ForDeclaration} \ [\text{?Yield}, \text{?Await}] \ \text{of} \ \text{AssignmentExpression} \ [+\text{In}, \text{?Yield}, \text{?Await}] \) \]
\( \text{Statement} \ [\text{?Yield}, \text{?Await}, \text{?Return}] \)

370
14.7.5.1 Static Semantics: Early Errors

ForInOfStatement :

- 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.

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

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).
2. Return ? ForIn/OfBodyEvaluation(ForBinding, Statement, keyResult, iterate, varBinding, labelSet).

ForInOfStatement : for ( ForDeclaration of AssignmentExpression ) Statement

1. Let keyResult be ? ForIn/OfHeadEvaluation(BoundNames of ForDeclaration, AssignmentExpression, iterate).
2. Return ? ForIn/OfBodyEvaluation(ForDeclaration, Statement, keyResult, iterate, lexicalBinding, labelSet).

ForInOfStatement : for await ( LeftHandSideExpression of AssignmentExpression ) Statement

1. Let keyResult be ? ForIn/OfHeadEvaluation(« », AssignmentExpression, async-iterate).
2. Return ? ForIn/OfBodyEvaluation(LeftHandSideExpression, Statement, keyResult, iterate, assignment, labelSet, async).

ForInOfStatement : for await ( var ForBinding of AssignmentExpression ) Statement

1. Let keyResult be ? ForIn/OfHeadEvaluation(« », AssignmentExpression, async-iterate).
2. Return ? ForIn/OfBodyEvaluation(ForBinding, Statement, keyResult, iterate, varBinding, 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:

1. Let oldEnv be the running execution context’s LexicalEnvironment.
2. If uninitializedBoundNames is not an empty List, then
   a. Assert: uninitializedBoundNames has no duplicate entries.
   b. Let newEnv be NewDeclarativeEnvironment(oldEnv).
   c. For each String name of uninitializedBoundNames, do
      i. Perform ! newEnv.CreateMutableBinding(name, false).
   d. Set the running execution context’s LexicalEnvironment to newEnv.
3. Let exprRef be the result of evaluating expr.
4. Set the running execution context’s LexicalEnvironment to oldEnv.
5. Let exprValue be ? GetValue(exprRef).
6. If iterationKind is enumerate, then
   a. If exprValue is undefined or null, then
      i. Return Completion { [[Type]]: break, [[Value]]: empty, [[Target]]: empty }.
   b. Let obj be ! ToObject(exprValue).
   c. Let iterator be ? EnumerateObjectProperties(obj).
   d. Let nextMethod be ! GetV(iterator, "next").
   e. Return the Record { [[Iterator]]: iterator, [[NextMethod]]: nextMethod, [[Done]]: false }.
7. 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.

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.

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).

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:

```javascript
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

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 behavior of \texttt{EnumerateObjectProperties}.

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 behavior of \texttt{EnumerateObjectProperties}.

14.7.5.10.1 Create\texttt{ForInIterator} (\texttt{object})

The abstract operation \texttt{CreateForInIterator} takes argument \texttt{object}. It is used to create a For-In Iterator object which iterates over the own and inherited enumerable string properties of \texttt{object} in a specific order. It performs the following steps when called:

1. \texttt{Assert: Type(object)} is Object.
2. Let \texttt{iterator} be \texttt{! OrdinaryObjectCreate(%ForInIteratorPrototype%, « [[Object]], [[ObjectWasVisited]], [[VisitedKeys]], [[RemainingKeys]] »)}.
3. Set \texttt{iterator.[[Object]]} to \texttt{object}.
4. Set \texttt{iterator.[[ObjectWasVisited]]} to false.
5. Set \texttt{iterator.[[VisitedKeys]]} to a new empty List.
6. Set \texttt{iterator.[[RemainingKeys]]} to a new empty List.
7. Return \texttt{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 \texttt{O} be the this value.
2. \texttt{Assert: Type(O)} is Object.
3. \texttt{Assert: O} has all of the internal slots of a For-In Iterator Instance (14.7.5.10.3).
4. Let \texttt{object} be \texttt{O.[[Object]]}.
5. Let \texttt{visited} be \texttt{O.[[VisitedKeys]]}.
6. Let \texttt{remaining} be \texttt{O.[[RemainingKeys]]}.
7. Repeat,
   a. If \texttt{O.[[ObjectWasVisited]]} is false, then
      i. Let \texttt{keys} be ? \texttt{object.[[OwnPropertyKeys]]}().
      ii. For each element \texttt{key} of \texttt{keys}, do
          1. If \texttt{Type(key)} is String, then
             a. Append \texttt{key} to \texttt{remaining}.

iii. Set \( O.\text{ObjectWasVisited} \) to \texttt{true}.

b. Repeat, while \texttt{remaining} is not empty,
   i. Let \( r \) be the first element of \texttt{remaining}.
   ii. Remove the first element from \texttt{remaining}.
   iii. If there does not exist an element \( v \) of \texttt{visited} such that \( \text{SameValue}(r, v) \) is \texttt{true}, then
      1. Let \( \text{desc} \) be \( \text{object.OwnProperty}(r) \).
      2. If \( \text{desc} \) is not \texttt{undefined}, then
         a. Append \( r \) to \texttt{visited}.
         b. If \( \text{desc.Enumerable} \) is \texttt{true}, return \texttt{CreateIterResultObject}(r, \texttt{false}).
   c. Set \texttt{object} to \( \text{object.PrototypeOf}() \).
   d. Set \( O.\text{Object} \) to \texttt{object}.
   e. Set \( O.\text{ObjectWasVisited} \) to \texttt{false}.
   f. If \texttt{object} is \texttt{null}, return \texttt{CreateIterResultObject}(\texttt{undefined}, \texttt{true}).

14.7.5.10.3 Properties of For-In Iterator Instances

For-In Iterator instances are ordinary objects that inherit properties from the \%\texttt{ForInIteratorPrototype}% intrinsic object. For-In Iterator instances are initially created with the internal slots listed in Table 38.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Object]]</td>
<td>The Object value whose properties are being iterated.</td>
</tr>
<tr>
<td>[[ObjectWasVisited]]</td>
<td>\texttt{true} if the iterator has invoked [[OwnPropertyKeys]] on [[Object]], \texttt{false} otherwise.</td>
</tr>
<tr>
<td>[[VisitedKeys]]</td>
<td>A list of String values which have been emitted by this iterator thus far.</td>
</tr>
<tr>
<td>[[RemainingKeys]]</td>
<td>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 \texttt{null}).</td>
</tr>
</tbody>
</table>

14.8 The \texttt{continue} Statement

Syntax

\[
\text{ContinueStatement}[\text{Yield, Await}] : \\
\text{continue} ; \\
\text{continue} \texttt{[no LineTerminator here]} \texttt{LabelIdentifier}[\texttt{?Yield, ?Await}] ;
\]

14.8.1 Static Semantics: Early Errors

\texttt{ContinueStatement} :

\[
\text{continue} ; \\
\text{continue} \texttt{LabelIdentifier} ;
\]

- It is a Syntax Error if this \texttt{ContinueStatement} is not nested, directly or indirectly (but not crossing function boundaries), within an \texttt{IterationStatement}.
14.8.2 Runtime Semantics: Evaluation

\[\begin{align*}
\text{ContinueStatement} & : \text{continue} ; \\
1. & \text{Return Completion} \{ [[\text{Type}]]: \text{continue}, [[\text{Value}]]: \text{empty}, [[\text{Target}]]: \text{empty} \}. \\
\text{ContinueStatement} & : \text{continue LabelIdentifier} ; \\
1. & \text{Let label be the StringValue of LabelIdentifier.} \\
2. & \text{Return Completion} \{ [[\text{Type}]]: \text{continue}, [[\text{Value}]]: \text{empty}, [[\text{Target}]]: label \}.
\end{align*}\]

14.9 The break Statement

Syntax

\[\text{BreakStatement[Yield, Await]} : \begin{align*}
\text{break} ; \\
\text{break [no LineTerminator here]} \text{LabelIdentifier[Yield, Await]} ;
\end{align*}\]

14.9.1 Static Semantics: Early Errors

\[\text{BreakStatement : break ;} \]

- It is a Syntax Error if this \text{BreakStatement} is not nested, directly or indirectly (but not crossing function boundaries), within an \text{IterationStatement} or a \text{SwitchStatement}.

14.9.2 Runtime Semantics: Evaluation

\[\text{BreakStatement : break ;} \]

1. Return Completion \{ [[\text{Type}]]: break, [[\text{Value}]]: empty, [[\text{Target}]]: empty \}.

\[\text{BreakStatement : break LabelIdentifier ;} \]

1. Let label be the StringValue of LabelIdentifier.
2. Return Completion \{ [[\text{Type}]]: break, [[\text{Value}]]: empty, [[\text{Target}]]: label \}.

14.10 The return Statement

Syntax

\[\text{ReturnStatement[Yield, Await]} : \begin{align*}
\text{return} ; \\
\text{return [no LineTerminator here]} \text{Expression[+In, Yield, Await]} ;
\end{align*}\]
14.10.1 Runtime Semantics: Evaluation

\[ \text{ReturnStatement} : \text{return} ; \]

1. Return \text{Completion} \{ [[Type]]: \text{return}, [[Value]]: \text{undefined}, [[Target]]: \text{empty} \}.

\[ \text{ReturnStatement} : \text{return} \text{Expression} ; \]

1. Let \text{exprRef} be the result of evaluating \text{Expression}.
2. Let \text{exprValue} be ? \text{GetValue}(\text{exprRef}).
3. If ! \text{GetGeneratorKind()} is \text{async}, set \text{exprValue} to ? \text{Await}(\text{exprValue}).
4. Return \text{Completion} \{ [[Type]]: \text{return}, [[Value]]: \text{exprValue}, [[Target]]: \text{empty} \}.

14.11 The with Statement

Syntax

\[ \text{WithStatement}[\text{Yield}, \text{Await}, \text{Return}] : \]

\[ \text{with ( Expression }[\text{+In}, ?\text{Yield}, ?\text{Await}) \text{ Statement}[?\text{Yield}, ?\text{Await}, ?\text{Return}] \]

The \text{with} statement adds an \text{object Environment Record} for a computed object to the lexical environment of the \text{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

\[ \text{WithStatement} : \text{with ( Expression ) Statement} \]

- It is a Syntax Error if the code that matches this production is contained in \text{strict mode code}.
- It is a Syntax Error if \text{IsLabelledFunction(Statement)} is \text{true}.

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

\[ \text{WithStatement} : \text{with ( Expression ) Statement} \]

1. Let \text{val} be the result of evaluating \text{Expression}.
2. Let \text{obj} be ? \text{ToObject(? GetValue(val))}.
3. Let \text{oldEnv} be the \text{running execution context}'s \text{LexicalEnvironment}.
4. Let $\text{newEnv}$ be $\text{NewObjectEnvironment}(\text{obj}, \text{oldEnv})$.
5. Set the $\text{withEnvironment}$ flag of $\text{newEnv}$ to $\text{true}$.
6. Set the running execution context's LexicalEnvironment to $\text{newEnv}$.
7. Let $C$ be the result of evaluating $\text{Statement}$.
8. Set the running execution context's LexicalEnvironment to $\text{oldEnv}$.
9. Return $\text{Completion}(\text{UpdateEmpty}(C, \text{undefined}))$.

NOTE

No matter how control leaves the embedded $\text{Statement}$, 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

$\text{SwitchStatement}[\text{Yield, Await, Return}] :$

switch ( $\text{Expression}[+\text{In}, ?\text{Yield}, ?\text{Await}]$ ) $\text{CaseBlock}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$

$\text{CaseBlock}[\text{Yield, Await, Return}] :$

{ $\text{CaseClauses}[?\text{Yield}, ?\text{Await}, ?\text{Return} \text{ opt}]$ }

{ $\text{CaseClauses}[?\text{Yield}, ?\text{Await}, ?\text{Return} \text{ opt}]$ $\text{DefaultClause}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$ }

$\text{CaseClauses}[\text{Yield, Await, Return}] :$

$\text{CaseClause}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$

$\text{CaseClauses}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$ $\text{CaseClause}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$

$\text{CaseClause}[\text{Yield, Await, Return}] :$

case $\text{Expression}[+\text{In}, ?\text{Yield}, ?\text{Await}]$ : $\text{StatementList}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$ opt

$\text{DefaultClause}[\text{Yield, Await, Return}] :$

default : $\text{StatementList}[?\text{Yield}, ?\text{Await}, ?\text{Return}]$ opt

14.12.1 Static Semantics: Early Errors

$\text{SwitchStatement} : \text{switch ( Expression ) CaseBlock}$

- It is a Syntax Error if the $\text{LexicallyDeclaredNames}$ of $\text{CaseBlock}$ contains any duplicate entries.
- It is a Syntax Error if any element of the $\text{LexicallyDeclaredNames}$ of $\text{CaseBlock}$ also occurs in the $\text{VarDeclaredNames}$ of $\text{CaseBlock}$.

14.12.2 Runtime Semantics: CaseBlockEvaluation

With parameter $\text{input}$.

$\text{CaseBlock} : \{ \}$

1. Return $\text{NormalCompletion}(\text{undefined})$. 
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 $R.\text{[Value]}$ is not empty, set $V$ to $R.\text{[Value]}$.
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 $: \text{case Expression} : \text{StatementList opt}$.
2. Let $exprRef$ be the result of evaluating the Expression of $C$.
3. Let clauseSelector be $? \text{GetValue}(exprRef)$.
4. Return the result of performing Strict Equality Comparison $input === \text{clauseSelector}$.

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 $: \text{switch ( Expression ) CaseBlock}$

1. Let $exprRef$ be the result of evaluating $Expression$.
2. Let $switchValue$ be $? \text{GetValue}(exprRef)$.
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 $oldEnv$.

NOTE No matter how control leaves the SwitchStatement the LexicalEnvironment is always restored to its former state.

CaseClause $: \text{case Expression :}$

1. Return NormalCompletion($empty$).

CaseClause $: \text{case Expression : StatementList}$

1. Return the result of evaluating StatementList.

DefaultClause $: \text{default :}$

1. Return NormalCompletion($empty$).

DefaultClause $: \text{default : StatementList}$

1. Return the result of evaluating StatementList.
14.13 Labelled Statements

Syntax

\[
\text{LabelledStatement} \quad \text{[Yield, Await, Return] :}
\]
\[
\text{LabelIdentifier} \quad \text{[?Yield, ?Await] : LabelledItem} \quad \text{[?Yield, ?Await, ?Return]}
\]

\[
\text{LabelledItem} \quad \text{[Yield, Await, Return] :}
\]
\[
\text{Statement} \quad \text{[?Yield, ?Await, ?Return]}
\]
\[
\text{FunctionDeclaration} \quad \text{[?Yield, ?Await, ~Default]}
\]

NOTE

A \textit{Statement} may be prefixed by a label. Labelled statements are only used in conjunction with labelled \texttt{break} and \texttt{continue} statements. ECMAScript has no \texttt{goto} statement. A \textit{Statement} can be part of a \textit{LabelledStatement}, which itself can be part of a \textit{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

\textit{LabelledItem} : \textit{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: \texttt{IsLabelledFunction (stmt)}

The abstract operation \texttt{IsLabelledFunction} takes argument \textit{stmt}. It performs the following steps when called:

1. If \textit{stmt} is not a \textit{LabelledStatement}, return \texttt{false}.
2. Let \textit{item} be the \textit{LabelledItem} of \textit{stmt}.
3. If \textit{item} is \textit{LabelledItem} : \textit{FunctionDeclaration}, return \texttt{true}.
4. Let \textit{subStmt} be the \textit{Statement} of \textit{item}.
5. Return \texttt{IsLabelledFunction(subStmt)}.

14.13.3 Runtime Semantics: Evaluation

\textit{LabelledStatement} : \textit{LabelIdentifier} : \textit{LabelledItem}

1. Let \textit{newLabelSet} be a new empty List.
2. Return \textit{LabelledEvaluation} of this \textit{LabelledStatement} with argument \textit{newLabelSet}.

14.13.4 Runtime Semantics: \texttt{LabelledEvaluation}

With parameter \textit{labelSet}.

\textit{BreakableStatement} : \textit{IterationStatement}
1. Let \(\text{stmtResult}\) be \(\text{LoopEvaluation}\) of \(\text{IterationStatement}\) with argument \(\text{labelSet}\).
2. If \(\text{stmtResult}.[[\text{Type}]]\) is \(\text{break}\), then
   a. If \(\text{stmtResult}.[[\text{Target}]]\) is \(\text{empty}\), then
      i. If \(\text{stmtResult}.[[\text{Value}]]\) is \(\text{empty}\), set \(\text{stmtResult}\) to \(\text{NormalCompletion(undefined)}\).
      ii. Else, set \(\text{stmtResult}\) to \(\text{NormalCompletion(stmtResult}.[[\text{Value}]]\))
3. Return \(\text{Completion(stmtResult)}\).

BreakableStatement : SwitchStatement

1. Let \(\text{stmtResult}\) be the result of evaluating \(\text{SwitchStatement}\).
2. If \(\text{stmtResult}.[[\text{Type}]]\) is \(\text{break}\), then
   a. If \(\text{stmtResult}.[[\text{Target}]]\) is \(\text{empty}\), then
      i. If \(\text{stmtResult}.[[\text{Value}]]\) is \(\text{empty}\), set \(\text{stmtResult}\) to \(\text{NormalCompletion(undefined)}\).
      ii. Else, set \(\text{stmtResult}\) to \(\text{NormalCompletion(stmtResult}.[[\text{Value}]]\))
3. Return \(\text{Completion(stmtResult)}\).

NOTE 1  A \(\text{BreakableStatement}\) is one that can be exited via an unlabelled \(\text{BreakStatement}\).

LabelledStatement : LabelIdentifier : LabelledItem

1. Let \(\text{label}\) be the \(\text{StringValue}\) of \(\text{LabelIdentifier}\).
2. Append \(\text{label}\) as an element of \(\text{labelSet}\).
3. Let \(\text{stmtResult}\) be \(\text{LabelledEvaluation}\) of \(\text{LabelledItem}\) with argument \(\text{labelSet}\).
4. If \(\text{stmtResult}.[[\text{Type}]]\) is \(\text{break}\) and \(\text{SameValue(stmtResult}.[[\text{Target}]], \text{label})\) is \(\text{true}\), then
   a. Set \(\text{stmtResult}\) to \(\text{NormalCompletion(stmtResult}.[[\text{Value}]]\))
5. Return \(\text{Completion(stmtResult)}\).

LabelledItem : FunctionDeclaration

1. Return the result of evaluating \(\text{FunctionDeclaration}\).

Statement : 
- \(\text{BlockStatement}\)
- \(\text{VariableStatement}\)
- \(\text{EmptyStatement}\)
- \(\text{ExpressionStatement}\)
- \(\text{IfStatement}\)
- \(\text{ContinueStatement}\)
- \(\text{BreakStatement}\)
- \(\text{ReturnStatement}\)
- \(\text{WithStatement}\)
- \(\text{ThrowStatement}\)
- \(\text{TryStatement}\)
- \(\text{DebuggerStatement}\)

1. Return the result of evaluating \(\text{Statement}\).
NOTE 2  The only two productions of Statement which have special semantics for LabelledEvaluation are BreakableStatement and LabelledStatement.

14.14 The throw Statement

Syntax

\[
\text{ThrowStatement} \rightarrow [\text{Yield}, \text{Await}] :\quad \text{throw} \ [\text{no LineTerminator here}] \ \text{Expression} \ [+\text{In}, \ ?\text{Yield}, \ ?\text{Await}] ;
\]


\[
\text{ThrowStatement} : \text{throw} \ \text{Expression} ;
\]

1. Let \( \text{exprRef} \) be the result of evaluating \( \text{Expression} \).
2. Let \( \text{exprValue} \) be \( ? \text{GetValue}(\text{exprRef}) \).
3. Return ThrowCompletion(\( \text{exprValue} \)).

14.15 The try Statement

Syntax

\[
\text{TryStatement} \rightarrow [\text{Yield}, \text{Await}, \text{Return}] :\quad \text{try} \ \text{Block} \[?\text{Yield}, \ ?\text{Await}, \ ?\text{Return}] \ \text{Catch}\[?\text{Yield}, \ ?\text{Await}, \ ?\text{Return}] \ \text{Finally}[?\text{Yield}, \ ?\text{Await}, \ ?\text{Return}]
\]

\[
\text{Catch} \rightarrow [\text{Yield}, \text{Await}, \text{Return}] :\quad \text{catch} ( \text{CatchParameter}[?\text{Yield}, \ ?\text{Await}] ) \ \text{Block}?[?\text{Yield}, \ ?\text{Await}, \ ?\text{Return}]
\]

\[
\text{Finally}[\text{Yield}, \text{Await}, \text{Return}] :\quad \text{finally} \ \text{Block}[?\text{Yield}, \ ?\text{Await}, \ ?\text{Return}]
\]

\[
\text{CatchParameter} \rightarrow [\text{Yield}, \text{Await}] :\quad \text{BindingIdentifier}[?\text{Yield}, \ ?\text{Await}]
\]

\[
\text{BindingPattern}[?\text{Yield}, \ ?\text{Await}]
\]

NOTE  The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exception-handling code. When a catch clause catches an exception, its CatchParameter is bound to that exception.

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 \( \text{Completion}(\text{UpdateEmpty}(F, \text{undefined})) \).

**TryStatement** : \texttt{try Block Catch Finally}

1. Let \( B \) be the result of evaluating \( \text{Block} \).
2. If \( B.\[Type]\) is \text{throw}, let \( C \) be \( \text{CatchClauseEvaluation} \) of \( \text{Catch} \) with argument \( B.\[Value] \).
3. Else, let \( C \) be \( B \).
4. Let \( F \) be the result of evaluating \( \text{Finally} \).
5. If \( F.\[Type]\) is normal, set \( F \) to \( C \).
6. Return \( \text{Completion}(\text{UpdateEmpty}(F, \text{undefined})) \).

### 14.16 The **debugger** Statement

**Syntax**

\[
\text{DebuggerStatement} := \text{debugger} \; ;
\]

**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** : \texttt{debugger} ;

1. If an \text{implementation-defined} debugging facility is available and enabled, then
   a. Perform an \text{implementation-defined} debugging action.
   b. Let \( \text{result} \) be an \text{implementation-defined Completion} value.
2. Else,
   a. Let \( \text{result} \) be \( \text{NormalCompletion}(\text{empty}) \).
3. Return \( \text{result} \).

### 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 \[[\text{Call}]\] internal method (10.2.1).

**15.1 Parameter Lists**

**Syntax**

\[
\text{UniqueFormalParameters}[\text{Yield, Await}] :=
\]

389
FormalParameters: [?Yield, ?Await]

FormalParameters: [Yield, Await]:
  [empty]
FunctionRestParameter: [Yield, ?Await]
FormalParameterList: [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 ]

  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.
1. Return false.

**FormalParameters**: [empty]

1. Return true.

**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.

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
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.

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] :

15.2.1 Static Semantics: Early Errors

FunctionDeclaration :
    function BindingIdentifier ( FormalParameters ) { FunctionBody }
    function ( FormalParameters ) { FunctionBody }

FunctionExpression :
    function BindingIdentifier opt ( 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

  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.
1. Assert: *name* is not present.
2. Set *name* to the *StringValue* of the *BindingIdentifier*.
3. Let *scope* be the running execution context’s *LexicalEnvironment*.
4. Let *funcEnv* be the *NewDeclarativeEnvironment*(*scope*).
5. Perform *funcEnv*.CreateImmutableBinding(*name*, false).
6. Let *sourceText* be the source text matched by the *FunctionExpression*.
7. Let *closure* be the *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 \[ \text{In, Yield, Await} \] : 
  ArrowParameters \[ ?\text{Yield, } ?\text{Await} \] \[ \text{no LineTerminator here} \] => ConciseBody \[ ?\text{In} \]

ArrowParameters \[ \text{Yield, Await} \] : 
  BindingIdentifier \[ ?\text{Yield, } ?\text{Await} \]
  CoverParenthesizedExpressionAndArrowParameterList \[ ?\text{Yield, } ?\text{Await} \]

ConciseBody \[ \text{In} \] : 
  \[ \text{lookahead} \neq \{ \} \] ExpressionBody \[ ?\text{In, } \sim\text{Await} \]
  \{ FunctionBody \[ \sim\text{Yield, } \sim\text{Await} \] \}

ExpressionBody \[ \text{In, Await} \] : 
  AssignmentExpression \[ ?\text{In, } \sim\text{Yield, } ?\text{Await} \]

### Supplemental Syntax

When processing an instance of the production

\[
\text{ArrowParameters} \[ \text{Yield, Await} \] : \text{CoverParenthesizedExpressionAndArrowParameterList} \[ ?\text{Yield, } ?\text{Await} \]
\]

the interpretation of \text{CoverParenthesizedExpressionAndArrowParameterList} is refined using the following grammar:

\[
\text{ArrowFormalParameters} \[ \text{Yield, Await} \] : 
  ( \text{UniqueFormalParameters} \[ ?\text{Yield, } ?\text{Await} \] )
\]

#### 15.3.1 Static Semantics: Early Errors

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

- It is a Syntax Error if \text{ArrowParameters} \text{Contains} \text{YieldExpression} is \text{true}.
- It is a Syntax Error if \text{ArrowParameters} \text{Contains} \text{AwaitExpression} is \text{true}.
- It is a Syntax Error if \text{ConciseBodyContainsUseStrict} of \text{ConciseBody} is \text{true} and \text{IsSimpleParameterList} of \text{ArrowParameters} is \text{false}.
- It is a Syntax Error if any element of the \text{BoundNames} of \text{ArrowParameters} also occurs in the \text{LexicallyDeclaredNames} of \text{ConciseBody}.

\[
\text{ArrowParameters} : \text{CoverParenthesizedExpressionAndArrowParameterList}
\]

- It is a Syntax Error if \text{CoverParenthesizedExpressionAndArrowParameterList} is not covering an \text{ArrowFormalParameters}.
- All early error rules for \text{ArrowFormalParameters} and its derived productions also apply to \text{CoveredFormalsList} of \text{CoverParenthesizedExpressionAndArrowParameterList}.

#### 15.3.2 Static Semantics: ConciseBodyContainsUseStrict

\[
\text{ConciseBody} : \text{ExpressionBody}
\]

1. Return \text{false}.

\[
\text{ConciseBody} : \{ \text{FunctionBody} \}
\]

1. Return \text{FunctionBodyContainsUseStrict} of \text{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

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


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 \texttt{prototype} be \texttt{%Function.prototype\}. 
6. Let \texttt{sourceText} be the source text matched by \texttt{MethodDefinition}.
7. Let \texttt{closure} be \texttt{OrdinaryFunctionCreate(prototype, sourceText, UniqueFormalParameters, FunctionBody, non-lexical-this, scope)}.
8. Perform \texttt{MakeMethod(closure, object)}.
9. Return the Record \{ [[Key]]: \texttt{propKey}, [[Closure]]: \texttt{closure} \}.

### 15.4.5 Runtime Semantics: MethodDefinitionEvaluation

With parameters \texttt{object} and \texttt{enumerable}.

\texttt{MethodDefinition : PropertyName \ ( UniqueFormalParameters \ ) \ \{ \ FunctionBody \ \}}

1. Let \texttt{methodDef} be \texttt{? DefineMethod} of \texttt{MethodDefinition} with argument \texttt{object}.
2. Perform \texttt{SetFunctionName(methodDef:[[Closure]], methodDef:[[Key]])}.
3. Let \texttt{desc} be the PropertyDescriptor \{ [[Value]]: \texttt{methodDef}[[Closure]], [[Writable]]: \texttt{true}, [[Enumerable]]: \texttt{enumerable}, [[Configurable]]: \texttt{true} \}.
4. Return \texttt{? DefinePropertyOrThrow(object, methodDef[[Key]], desc)}.

\texttt{MethodDefinition : \texttt{get} \ PropertyName \ ( \ ) \ \{ \ FunctionBody \ \}}

1. Let \texttt{propKey} be the result of evaluating \texttt{PropertyName}.
2. ReturnIfAbrupt(\texttt{propKey}).
3. Let \texttt{scope} be the running execution context’s LexicalEnvironment.
4. Let \texttt{sourceText} be the source text matched by \texttt{MethodDefinition}.
5. Let \texttt{formalParameterList} be an instance of the production \texttt{FormalParameters : [empty]}.
6. Let \texttt{closure} be \texttt{OrdinaryFunctionCreate(%Function.prototype%, sourceText, formalParameterList, FunctionBody, non-lexical-this, scope)}.
7. Perform \texttt{MakeMethod(closure, object)}.
8. Perform \texttt{SetFunctionName(closure, propKey, "get")}.
9. Let \texttt{desc} be the PropertyDescriptor \{ [[Get]]: \texttt{closure}, [[Enumerable]]: \texttt{enumerable}, [[Configurable]]: \texttt{true} \}.
10. Return \texttt{? DefinePropertyOrThrow(object, propKey, desc)}.

\texttt{MethodDefinition : \texttt{set} \ PropertyName \ ( \ PropertySetParameterList \ ) \ \{ \ FunctionBody \ \}}

1. Let \texttt{propKey} be the result of evaluating \texttt{PropertyName}.
2. ReturnIfAbrupt(\texttt{propKey}).
3. Let \texttt{scope} be the running execution context’s LexicalEnvironment.
4. Let \texttt{sourceText} be the source text matched by \texttt{MethodDefinition}.
5. Let \texttt{closure} be \texttt{OrdinaryFunctionCreate(%Function.prototype%, sourceText, PropertySetParameterList, FunctionBody, non-lexical-this, scope)}.
6. Perform \texttt{MakeMethod(closure, object)}.
7. Perform \texttt{SetFunctionName(closure, propKey, "set")}.
8. Let \texttt{desc} be the PropertyDescriptor \{ [[Set]]: \texttt{closure}, [[Enumerable]]: \texttt{enumerable}, [[Configurable]]: \texttt{true} \}.
9. Return \texttt{? DefinePropertyOrThrow(object, propKey, desc)}.

\texttt{GeneratorMethod : \* \ PropertyName \ ( \ UniqueFormalParameters \ ) \ \{ \ GeneratorBody \ \}}

1. Let \texttt{propKey} be the result of evaluating \texttt{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).
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 }.

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).
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 }.

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).
7. Perform ! SetFunctionName(closure, propKey).
8. Let desc be the PropertyDescriptor { [[Value]]: closure, [[Writable]]: true, [[Enumerable]]: enumerable, [[Configurable]]: true }.

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]

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
• 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 GeneratorBody Contains SuperCall is true.

15.5.2 Runtime Semantics: EvaluateGeneratorBody

With parameters functionObject and argumentsList (a List).

GeneratorBody : FunctionBody

2. Let G be ? OrdinaryCreateFromConstructor(functionObject, "%GeneratorFunction.prototype.prototype%", « [[GeneratorState]], [[GeneratorContext]], [[GeneratorBrand]] »).
3. Set G.[[GeneratorBrand]] to empty.
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.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.prototype%, sourceText, FormalParameters, GeneratorBody, non-lexical-this, scope).
3. Perform SetFunctionName(F, "default").
4. Let prototype be ! OrdinaryObjectCreate(%GeneratorFunction.prototype.prototype.prototype%).
5. Perform `DefinePropertyOrThrow(F, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.

**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 })`.

**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


YieldExpression : yield AssignmentExpression

1. Let exprRef be the result of evaluating AssignmentExpression.
2. Let value be ? GetValue(exprRef).

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
      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
         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 \( \text{Completion} \{ \text{[Type]}: \text{normal}, \text{[Value]}: \text{empty}, \text{[Target]}: \text{empty} \} \).

3. If \( \text{generatorKind} \) is \( \text{async} \), perform \( \text{AsyncIteratorClose(iteratorRecord, closeCompletion)} \).

4. Else, perform \( \text{IteratorClose(iteratorRecord, closeCompletion)} \).

5. NOTE: The next step throws a \( \text{TypeError} \) to indicate that there was a \( \text{yield*} \) protocol violation: \( \text{iterator} \) does not have a \( \text{throw} \) method.

6. Throw a \( \text{TypeError} \) exception.

c. Else,
   i. \( \text{Assert}: \text{received}.\text{[Type]} \) is \( \text{return} \).
   ii. Let \( \text{return} \) be \( \text{GetMethod(iterator, "return")} \).
   iii. If \( \text{return} \) is \( \text{undefined} \), then
      1. If \( \text{generatorKind} \) is \( \text{async} \), set \( \text{received}.\text{[Value]} \) to \( \text{Await(received}.\text{[Value]}) \).
      2. Return \( \text{Completion(received)} \).
   iv. Let \( \text{innerReturnResult} \) be \( \text{Call(return, iterator, « received.\text{[Value]} »)} \).
   v. If \( \text{generatorKind} \) is \( \text{async} \), set \( \text{innerReturnResult} \) to \( \text{Await(innerReturnResult)} \).
   vi. If \( \text{Type(innerReturnResult)} \) is not \( \text{Object} \), throw a \( \text{TypeError} \) exception.
   vii. Let \( \text{done} \) be \( \text{IteratorComplete(innerReturnResult)} \).
   viii. If \( \text{done} \) is \( \text{true} \), then
      1. Let \( \text{value} \) be \( \text{IteratorValue(innerReturnResult)} \).
      2. Return \( \text{Completion} \{ \text{[Type]}: \text{return}, \text{[Value]}: \text{value}, \text{[Target]}: \text{empty} \} \).
   ix. If \( \text{generatorKind} \) is \( \text{async} \), set \( \text{received} \) to \( \text{AsyncGeneratorYield(? IteratorValue(innerReturnResult))} \).
   x. Else, set \( \text{received} \) to \( \text{GeneratorYield(innerReturnResult)} \).

### 15.6 Async Generator Function Definitions

**Syntax**

\[
\text{AsyncGeneratorMethod} \ \text{[Yield, Await]} \ :
\quad \text{async} \ [\text{no LineTerminator here}] * \text{PropertyName} \ \text{[?Yield, ?Await]} \ ( \leftarrow \text{UniqueFormalParameters[+Yield, +Await]} \ ) \ \{ \text{AsyncGeneratorBody} \ }
\]

\[
\text{AsyncGeneratorDeclaration} \ \text{[Yield, Await, Default]} \ :
\quad \text{async} \ [\text{no LineTerminator here}] \ \text{function} * \ \text{BindingIdentifier} \ \text{[?Yield, ?Await]} \ ( \leftarrow \text{FormalParameters[+Yield, +Await]} \ ) \ \{ \text{AsyncGeneratorBody} \ }
\]

\[
\quad [\text{+Default}] \ \text{async} \ [\text{no LineTerminator here}] \ \text{function} * \ ( \text{FormalParameters[+Yield, +Await]} \ ) \ \{ \text{AsyncGeneratorBody} \ }
\]

\[
\text{AsyncGeneratorExpression} : \quad \text{async} \ [\text{no LineTerminator here}] \ \text{function} * \ \text{BindingIdentifier} \ \text{[+Yield, +Await]} \ \text{opt} \ ( \leftarrow \text{FormalParameters[+Yield, +Await]} \ ) \ \{ \text{AsyncGeneratorBody} \ }
\]

\[
\text{AsyncGeneratorBody} : \quad \text{FunctionBody[+Yield, +Await]}
\]
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

2. Let generator be ? OrdinaryCreateFromConstructor(functionObject,
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.prototype%, sourceText, FormalParameters, AsyncGeneratorBody, non-lexical-this, scope)`.
4. Perform `! SetFunctionName(F, name)`.
5. Let `prototype` be `! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.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.prototype%, sourceText, FormalParameters, AsyncGeneratorBody, non-lexical-this, scope)`.
3. Perform `! SetFunctionName(F, "default")`.
4. Let `prototype` be `! OrdinaryObjectCreate(%AsyncGeneratorFunction.prototype.prototype.prototype%)`.
5. Perform `! DefinePropertyOrThrow(F, "prototype", PropertyDescriptor { [[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false })`.

**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.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 }).

**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 }
```
A class definition is always strict mode code.

### 15.7.1 Static Semantics: Early Errors

#### ClassTail

- 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`.

- It is a Syntax Error if `PrototypePropertyNameList` of `ClassElementList` contains more than one occurrence of "`constructor`".

#### ClassBody

- 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

- 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

- If `PropName` of `MethodDefinition` is "`constructor`", return `ConstructorMethod`.
- Return `NonConstructorMethod`.

1. Return `NonConstructorMethod`.

1. Return `empty`.

### 15.7.3 Static Semantics: ConstructorMethod

1. If `ClassElementKind` of `ClassElement` is `ConstructorMethod`, return `ClassElement`.
2. Return `empty`.

### 15.7.4 Static Semantics: IsStatic

1. Return `false`.

1. Return `true`.

1. Return `false`.

### 15.7.5 Static Semantics: NonConstructorMethodDefinitions

1. If `ClassElementKind` of `ClassElement` is `NonConstructorMethod`, then
   a. Return a `List` whose sole element is `ClassElement`.
2. Return a new empty `List`.

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`.

---

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.
15.7.6 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.
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.
15. If ClassBody_opt is not present, let methods be a new empty List.
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.
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."

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**

```javascript
AsyncFunctionDeclaration [Yield, Await, Default] :

[+Default] async [no LineTerminator here] function (FormalParameters [~Yield, +Await]) {}
```

416
AsyncFunctionExpression:

```javascript
async [no LineTerminator here] function BindingIdentifier[-Yield, +Await] opt (FormalParameters[-Yield, +Await]) { AsyncFunctionBody }
```

AsyncMethod [Yield, Await] :

```javascript
```

AsyncFunctionBody:

```javascript
FunctionBody[-Yield, +Await]
```

AwaitExpression [Yield] :

```javascript
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**:

```javascript
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 ( FormaParameters ) { AsyncFunctionBody }
async function ( FormaParameters ) { AsyncFunctionBody }

AsyncFunctionExpression :
  async function BindingIdentifier_opt ( FormaParameters ) { 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 ( FormaParameters ) { 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 ( FormaParameters ) { 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 ( FormaParameters ) { 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).

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).

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
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 }

`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] :
  AsyncConciseBody [Yield, Await]
CoverCallExpressionAndAsyncArrowHead [Yield, Await] [no LineTerminator here] =>
  AsyncConciseBody [Yield, Await]

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 :
```

### 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`.  

420
• 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` }`


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
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.
7. Perform SetFunctionName(closure, name).

15.9.6 Runtime Semantics: Evaluation
AsyncArrowFunction :
   async AsyncArrowBindingIdentifier => AsyncConciseBody
   CoverCallExpressionAndAsyncArrowHead => AsyncConciseBody


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.
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<sub>opt</sub>

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(undef).

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>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Realm]]</td>
<td>Realm Record * undefined</td>
<td>The realm within which this script was created. undefined if not yet assigned.</td>
</tr>
<tr>
<td>[[Environment]]</td>
<td>Environment Record * undefined</td>
<td>The Environment Record containing the top level bindings for this script. This field is set when the script is instantiated.</td>
</tr>
<tr>
<td>[[ECMAScriptCode]]</td>
<td>a Parse Node</td>
<td>The result of parsing the source text of this script using Script as the goal symbol.</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is empty.</td>
<td>Field reserved for use by host environments that need to associate additional information with a script.</td>
</tr>
</tbody>
</table>
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
         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
             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 $env$.

17. For each String $vn$ of declaredVarNames, do

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 : ModuleBodyopt

ModuleBody : ModuleItemList

ModuleItemList : ModuleItem ModuleItemList

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 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>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Realm]]</td>
<td>Realm Record</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Environment]]</td>
<td>module Environment Record</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Namespace]]</td>
<td>Object</td>
<td>undefined</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is undefined</td>
<td></td>
</tr>
</tbody>
</table>
### Table 41: Abstract Methods of Module Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetExportedNames(exportStarSet)</td>
<td>Return a list of all names that are either directly or indirectly exported from this module.</td>
</tr>
<tr>
<td>ResolveExport(exportName [, resolveSet])</td>
<td>Return the binding of a name exported by this module. Bindings are represented by a ResolvedBinding Record, 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 &quot;<em>namespace</em>&quot;. Return null if the name cannot be resolved, or &quot;ambiguous&quot; if multiple bindings were found. Each time this operation is called with a specific exportName, resolveSet pair as arguments it must return the same result if it completes normally.</td>
</tr>
<tr>
<td>Link()</td>
<td>Prepare the module for evaluation by transitively resolving all module dependencies and creating a module Environment Record.</td>
</tr>
<tr>
<td>Evaluate()</td>
<td>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. Link must have completed successfully prior to invoking this method.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Status]]</td>
<td>unlinked, linking, linked, evaluating, evaluated</td>
<td>Initially unlinked. Transitions to linking, linked, evaluating, evaluated (in that order) as the module progresses throughout its lifecycle.</td>
</tr>
<tr>
<td>[[EvaluationError]]</td>
<td>abrupt completion, undefined</td>
<td>A completion of type throw representing the exception that occurred during evaluation. undefined if no exception occurred or if [[Status]] is not evaluated.</td>
</tr>
<tr>
<td>[[DFSIndex]]</td>
<td>Integer, undefined</td>
<td>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.</td>
</tr>
<tr>
<td>[[DFSAncestorIndex]]</td>
<td>Integer, undefined</td>
<td>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 &quot;earlier&quot; module in the same strongly connected component.</td>
</tr>
<tr>
<td>[[RequestedModules]]</td>
<td>List of String</td>
<td>A List of all the ModuleSpecifier strings used by the module represented by this record to request the importation of a module. The List is source code occurrence ordered.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeEnvironment()</td>
<td>Initialize the Environment Record of the module, including resolving all imported bindings, and create the module's execution context.</td>
</tr>
<tr>
<td>ExecuteModule()</td>
<td>Evaluate the module's code within its execution context.</td>
</tr>
</tbody>
</table>

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:** \( m.\text{[Status]} \) is linking.
ii. Set \( m.\text{[Status]} \) to unlinked.
iii. Set \( m.\text{[Environment]} \) to **undefined**.
iv. Set \( m.\text{[DFSIndex]} \) to **undefined**.
v. Set \( m.\text{[DFSAncestorIndex]} \) to **undefined**.

b. **Assert:** \( module.\text{[Status]} \) is unlinked.

c. Return result.

5. **Assert:** \( module.\text{[Status]} \) 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 \( \text{[DFSIndex]} \) and \( \text{[DFSAncestorIndex]} \) fields, keep track of the depth-first search (DFS) traversal. In particular, \( \text{[DFSAncestorIndex]} \) 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 ? \( module.\text{Link()} \).
   b. Return \( index \).
2. If \( module.\text{[Status]} \) is linking, linked, or evaluated, then
   a. Return \( index \).
3. **Assert:** \( module.\text{[Status]} \) is unlinked.
4. Set \( module.\text{[Status]} \) to linking.
5. Set \( module.\text{[DFSIndex]} \) to \( index \).
6. Set \( module.\text{[DFSAncestorIndex]} \) to \( index \).
7. Set \( index \) to \( index + 1 \).
8. Append \( module \) to \( stack \).
9. For each String \( \text{required} \) of \( module.\text{[RequestedModules]} \), do
   a. Let \( \text{requiredModule} \) be ? \( \text{HostResolveImportedModule}(module, \text{required}) \).
   b. Set \( index \) to ? \( \text{InnerModuleLinking}(\text{requiredModule}, stack, index) \).
   c. If \( \text{requiredModule} \) is a Cyclic Module Record, then
      i. **Assert:** \( \text{requiredModule}.\text{[Status]} \) is either linking, linked, or evaluated.
      ii. **Assert:** \( \text{requiredModule}.\text{[Status]} \) is linking if and only if \( \text{requiredModule} \) is in \( stack \).
      iii. If \( \text{requiredModule}.\text{[Status]} \) is linking, then
         1. Set \( module.\text{[DFSAncestorIndex]} \) to \( \text{min}(module.\text{[DFSAncestorIndex]}), \text{requiredModule}.\text{[DFSAncestorIndex]} \).
10. Perform ? \( module.\text{InitializeEnvironment}() \).
11. **Assert:** \( module \) occurs exactly once in \( stack \).
12. **Assert:** \( module.\text{[DFSAncestorIndex]} \leq module.\text{[DFSIndex]} \).
13. If \( module.\text{[DFSAncestorIndex]} = module.\text{[DFSIndex]} \), then
    a. Let \( done \) be false.
    b. Repeat, while \( done \) is false,
       i. Let \( \text{requiredModule} \) 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`.


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`. 438
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().
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:

![Diagram: 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

439
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

```
A
  ↓
???
```

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

```
A
  /\  /
 /  \ /  \
B    C
```
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 $A.\text{[[Status]]}$ is already linking. $B.\text{[[Status]]}$ itself remains linking when control gets back to $A$ and InnerModuleLinking is triggered on $C$. After this returns with $C.\text{[[Status]]}$ 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 a 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 $\text{[[EvaluationError]]}$ fields, while $C$ is left as evaluated with no $\text{[[EvaluationError]]}$.

### 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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[ECMAScriptCode]]</td>
<td>a Parse Node</td>
<td>The result of parsing the source text of this module using <code>Module</code> as the goal symbol.</td>
</tr>
<tr>
<td>[[Context]]</td>
<td>An ECMAScript execution context.</td>
<td>The execution context associated with this module.</td>
</tr>
<tr>
<td>[[ImportMeta]]</td>
<td>Object</td>
<td>An object exposed through the <code>import.meta</code> meta property. It is empty until it is accessed by ECMAScript code.</td>
</tr>
<tr>
<td>[[ImportEntries]]</td>
<td>List of ImportEntry Records</td>
<td>A List of ImportEntry records derived from the code of this module.</td>
</tr>
<tr>
<td>[[LocalExportEntries]]</td>
<td>List of ExportEntry Records</td>
<td>A List of ExportEntry records derived from the code of this module that correspond to declarations that occur within the module.</td>
</tr>
<tr>
<td>[[IndirectExportEntries]]</td>
<td>List of ExportEntry Records</td>
<td>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 <code>export * as namespace</code> declarations.</td>
</tr>
<tr>
<td>[[StarExportEntries]]</td>
<td>List of ExportEntry Records</td>
<td>A List of ExportEntry records derived from the code of this module that correspond to <code>export *</code> declarations that occur within the module, not including <code>export * as namespace</code> declarations.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[ModuleRequest]]</td>
<td>String</td>
<td>String value of the <code>ModuleSpecifier</code> of the <code>ImportDeclaration</code>.</td>
</tr>
<tr>
<td>[[ImportName]]</td>
<td>String</td>
<td>The name under which the desired binding is exported by the module identified by <code>[[ModuleRequest]]</code>. The value &quot;&quot;&quot;&quot; indicates that the import request is for the target module's namespace object.</td>
</tr>
<tr>
<td>[[LocalName]]</td>
<td>String</td>
<td>The name that is used to locally access the imported value from within the importing module.</td>
</tr>
</tbody>
</table>
Table 46 gives examples of ImportEntry records fields used to represent the syntactic import forms:

Table 46 (Informative): Import Forms Mappings to ImportEntry Records

<table>
<thead>
<tr>
<th>Import Statement Form</th>
<th>[[ModuleRequest]]</th>
<th>[[ImportName]]</th>
<th>[[LocalName]]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>import v from &quot;mod&quot;;</code></td>
<td>&quot;mod&quot;</td>
<td>&quot;default&quot;</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td><code>import * as ns from &quot;mod&quot;;</code></td>
<td>&quot;mod&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;ns&quot;</td>
</tr>
<tr>
<td><code>import {x} from &quot;mod&quot;;</code></td>
<td>&quot;mod&quot;</td>
<td>&quot;x&quot;</td>
<td>&quot;x&quot;</td>
</tr>
<tr>
<td><code>import {x as v} from &quot;mod&quot;;</code></td>
<td>&quot;mod&quot;</td>
<td>&quot;x&quot;</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td><code>import &quot;mod&quot;;</code></td>
<td>An ImportEntry Record is not created.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[ExportName]]</td>
<td>String</td>
<td>The name used to export this binding by this module.</td>
</tr>
<tr>
<td>[[ModuleRequest]]</td>
<td>String</td>
<td>The String value of the ModuleSpecifier of the ExportDeclaration. null if the ExportDeclaration does not have a ModuleSpecifier.</td>
</tr>
<tr>
<td>[[ImportName]]</td>
<td>String</td>
<td>The name under which the desired binding is exported by the module identified by [[ModuleRequest]]. null if the ExportDeclaration does not have a ModuleSpecifier. &quot;*&quot; indicates that the export request is for all exported bindings.</td>
</tr>
<tr>
<td>[[LocalName]]</td>
<td>String</td>
<td>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.</td>
</tr>
</tbody>
</table>
NOTE 2

Table 48 gives examples of the ExportEntry record fields used to represent the syntactic export forms:

<table>
<thead>
<tr>
<th>Export Statement Form</th>
<th>[[ExportName]]</th>
<th>[[ModuleRequest]]</th>
<th>[[ImportName]]</th>
<th>[[LocalName]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>export var v;</td>
<td>&quot;v&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td>export default function f() {}</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>export default function () {}</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;&quot;default&quot;&quot;</td>
</tr>
<tr>
<td>export default 42;</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;&quot;default&quot;&quot;</td>
</tr>
<tr>
<td>export {x};</td>
<td>&quot;x&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;x&quot;</td>
</tr>
<tr>
<td>export {v as x};</td>
<td>&quot;x&quot;</td>
<td>null</td>
<td>null</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td>export {x} from &quot;mod&quot;;</td>
<td>&quot;x&quot;</td>
<td>&quot;mod&quot;</td>
<td>&quot;x&quot;</td>
<td>null</td>
</tr>
<tr>
<td>export {v as x} from &quot;mod&quot;;</td>
<td>&quot;x&quot;</td>
<td>&quot;mod&quot;</td>
<td>&quot;v&quot;</td>
<td>null</td>
</tr>
<tr>
<td>export * from &quot;mod&quot;;</td>
<td>null</td>
<td>&quot;mod&quot;</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>export * as ns from &quot;mod&quot;;</td>
<td>&quot;ns&quot;</td>
<td>&quot;mod&quot;</td>
<td></td>
<td>null</td>
</tr>
</tbody>
</table>

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 the 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 the 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.
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.
   
   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
         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
      
      ii. Else,
      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

```html
<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
   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 `ResolvedBinding 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))`.
NOTE

1. Return \texttt{NormalCompletion}(\texttt{empty}).

16.2.2 Imports

Syntax

import \texttt{ImportClause FromClause ;}

\texttt{import ModuleSpecifier ;}

ImportClause :

ImportedDefaultBinding

NameSpaceImport

NamedImports

ImportedDefaultBinding , NameSpaceImport

ImportedDefaultBinding , NamedImports

ImportedDefaultBinding :

ImportedBinding

NameSpaceImport :

\texttt{* as ImportedBinding}

NamedImports :

\{ \}

\{ \texttt{ImportsList} \}

\{ \texttt{ImportsList} , \}

FromClause :

\texttt{from ModuleSpecifier}

ImportsList :

ImportSpecifier

ImportsList , ImportSpecifier

ImportSpecifier :

ImportedBinding

IdentifierName \texttt{as ImportedBinding}

ModuleSpecifier :

StringLiteral

ImportedBinding :

\texttt{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
  StatementList

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
```
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 \( \text{names} \) be ExportedBindings of ModuleItemList.
2. Append to \( \text{names} \) the elements of the ExportedBindings of ModuleItem.
3. Return \( \text{names} \).

ModuleItem :

\[
\begin{align*}
\text{ImportDeclaration} \\
\text{StatementListItem}
\end{align*}
\]

1. Return a new empty List.

ExportDeclaration :

\[
\begin{align*}
\text{export} & \ \text{ExportFromClause} \ \text{FromClause} \\
& ;
\end{align*}
\]

1. Return a new empty List.

ExportDeclaration : export NamedExports ;

1. Return the ExportedBindings of NamedExports.
ExportDeclaration : \texttt{export} VariableStatement

1. Return the BoundNames of VariableStatement.

ExportDeclaration : \texttt{export} Declaration

1. Return the BoundNames of Declaration.

ExportDeclaration :
\begin{itemize}
  \item \texttt{export default} HoistableDeclaration
  \item \texttt{export default} ClassDeclaration
  \item \texttt{export default} AssignmentExpression ;
\end{itemize}

1. Return the BoundNames of this ExportDeclaration.

NamedExports : \{ \}

1. Return a new empty List.

ExportsList : ExportsList , ExportSpecifier

1. Let \textit{names} be the ExportedBindings of ExportsList.
2. Append to \textit{names} the elements of the ExportedBindings of ExportSpecifier.
3. Return \textit{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.

\subsection*{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 \textit{names} be ExportedNames of ModuleItemList.
2. Append to \textit{names} the elements of the ExportedNames of ModuleItem.
3. Return \textit{names}.

ModuleItem : ExportDeclaration

1. Return the ExportedNames of ExportDeclaration.

ModuleItem :
\begin{itemize}
  \item ImportDeclaration
  \item StatementListItem
\end{itemize}
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 : exportDeclaration

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" }.
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.
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 1F.

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 +∞ (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 }.
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()`.
   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()].
      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`.

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
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 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 \textit{varNames} be the \textit{VarDeclaredNames} of \textit{body}.
2. Let \textit{varDeclarations} be the \textit{VarScopedDeclarations} of \textit{body}.
3. If \textit{strict} is \textit{false}, then
   a. If \textit{varEnv} is a global Environment Record, then
      i. For each element \textit{name} of \textit{varNames}, do
         1. If \textit{varEnv.HasLexicalDeclaration} (\textit{name}) is \textit{true}, throw a SyntaxError exception.
         2. NOTE: \texttt{eval eval} will not create a global var declaration that would be shadowed by a global lexical declaration.
   b. Let \textit{thisEnv} be lexEnv.
   c. Assert: The following loop will terminate.
   d. Repeat, while \textit{thisEnv} is not the same as \textit{varEnv},
      i. If \textit{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 \textit{name} of \textit{varNames}, do
             a. If \textit{thisEnv.HasBinding} (\textit{name}) is \textit{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 \textit{thisEnv} to \textit{thisEnv}.[[OuterEnv]].
4. Let \textit{functionsToInitialize} be a new empty List.
5. Let \textit{declaredFunctionNames} be a new empty List.
6. For each element \textit{d} of \textit{varDeclarations}, in reverse List order, do
   a. If \textit{d} is neither a VariableDeclaration nor a ForBinding nor a BindingIdentifier, then
      i. Assert: \textit{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 \textit{fn} be the sole element of the BoundNames of \textit{d}.
   iv. If \textit{fn} is not an element of \textit{declaredFunctionNames}, then
      1. If \textit{varEnv} is a global Environment Record, then
         a. Let \textit{fnDefinable} be \texttt{? varEnv.CanDeclareGlobalFunction(fn)}.
         b. If \textit{fnDefinable} is \textit{false}, throw a TypeError exception.
      2. Append \textit{fn} to \textit{declaredFunctionNames}.
      3. Insert \textit{d} as the first element of \textit{functionsToInitialize}.
7. NOTE: Annex B.3.3.3 adds additional steps at this point.
8. Let \textit{declaredVarNames} be a new empty List.
9. For each element \textit{d} of \textit{varDeclarations}, do
   a. If \textit{d} is a VariableDeclaration, a ForBinding, or a BindingIdentifier, then
      i. For each String \textit{vn} of the BoundNames of \textit{d}, do
         1. If \textit{vn} is not an element of \textit{declaredFunctionNames}, then
            a. If \textit{varEnv} is a global Environment Record, then
               i. Let \textit{vnDefinable} be \texttt{? varEnv.CanDeclareGlobalVar(vn)}.
               ii. If \textit{vnDefinable} is \textit{false}, throw a TypeError exception.
            b. If \textit{vn} is not an element of \textit{declaredVarNames}, then
               i. Append \textit{vn} to \textit{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, \(+\infty\text{F}_\text{F}\) or \(-\infty\text{F}_\text{F}\), 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.
   b. Return +0.
7. Return ❄(`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 \( \text{sign} \) 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 \( \text{stripPrefix} \) be \text{true}.

8. If \( R \neq 0 \), then
   a. If \( R < 2 \) or \( R > 36 \), return \( \text{NaN} \).
   b. If \( R 
eq 16 \), set \( \text{stripPrefix} \) to \text{false}.

9. Else,
   a. Set \( R \) to 10.

10. If \( \text{stripPrefix} \) is \text{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 \( \text{end} \) be the index within \( S \) of the first such code unit; otherwise, let \( \text{end} \) be the length of \( S \).

12. Let \( Z \) be the substring of \( S \) from 0 to \( \text{end} \).

13. If \( Z \) is empty, return \( \text{NaN} \).

14. Let \( \text{mathInt} \) be the \text{integer} value that is represented by \( Z \) in radix-\( R \) notation, using the letters \( \text{A-Z} \) and \( \text{a-z} \) 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 \( \text{mathInt} \) may be an \text{implementation-approximated} value representing the \text{integer} value that is represented by \( Z \) in radix-\( R \) notation.)

15. If \( \text{mathInt} = 0 \), then
    a. If \( \text{sign} = -1 \), return \(-0_\mathbb{R}\).
    b. Return \(+0_\mathbb{R}\).

16. Return \( \mathbb{R}(\text{sign} \times \text{mathInt}) \).

\textbf{NOTE} \hspace{1cm} \text{parseInt} may interpret only a leading portion of \text{string} as an \text{integer} value; it ignores any code units that cannot be interpreted as part of the notation of an \text{integer}, and no indication is given that any such code units were ignored.

\textbf{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 \textbf{19.2.6.2}, \textbf{19.2.6.3}, \textbf{19.2.6.4} and \textbf{19.2.6.5}

\textbf{NOTE} \hspace{1cm} Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

\textbf{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:
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 \(k = strLen\), 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 \(\text{CodePointAt}(string, k)\).
      ii. If \(cp.\text{[[IsUnpairedSurrogate]]}\) is \(true\), throw a URIError exception.
      iii. Set \(k\) to \(k + cp.\text{[[CodeUnitCount]]}\).
      iv. Let \(Octets\) be the List of octets resulting by applying the UTF-8 transformation to \(cp.\text{[[CodePoint]]}\).
      v. For each element \(octet\) of \(Octets\), do
         1. Set \(R\) to the string-concatenation of:
            i. \(R\)
            ii. "\%"
            iii. 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 \(k = strLen\), 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 \(k + 2 \geq strLen\), 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 $\text{reservedSet}$, then
      a. Let $S$ be the String value containing only the code unit $C$.
   3. Else,
      a. Let $S$ be the substring of $\text{string}$ from $\text{start}$ to $k + 1$.
viii. Else,
   1. If $n = 1$ or $n > 4$, throw a $\text{URIError}$ exception.
   2. If $k + (3 \times (n - 1)) \geq \text{strLen}$, throw a $\text{URIError}$ exception.
   3. Let $\text{Octets}$ 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 $\text{string}$ is not the code unit 0x0025 (PERCENT SIGN), throw a $\text{URIError}$ exception.
      c. If the code units at index $(k + 1)$ and $(k + 2)$ within $\text{string}$ do not represent hexadecimal digits, throw a $\text{URIError}$ 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 $\text{Octets}$.
      g. Set $j$ to $j + 1$.
   6. Assert: The length of $\text{Octets}$ is $n$.
   7. If $\text{Octets}$ does not contain a valid UTF-8 encoding of a Unicode code point, throw a $\text{URIError}$ exception.
   8. Let $V$ be the code point obtained by applying the UTF-8 transformation to $\text{Octets}$, that is, from a List of octets into a 21-bit value.
   9. Let $S$ be $\text{UTF16EncodeCodePoint}(V)$.
   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.
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. Escape sequences that could not have been introduced by `encodeURIComponent` are not replaced.

<table>
<thead>
<tr>
<th>Code Unit Value</th>
<th>Representation</th>
<th>1st Octet</th>
<th>2nd Octet</th>
<th>3rd Octet</th>
<th>4th Octet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 - 0x007F</td>
<td>00000000 0zzzzzzz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0080 - 0x00FF</td>
<td>00000yyyy yyzzzzzz</td>
<td>110yyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0xD7FF</td>
<td>xxxxyyyyy yyzzzzzz</td>
<td>1110xxx 10yyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xD800 - 0xDBFF</td>
<td>110110vv vyyyyyyyyyy</td>
<td>11110uuu 10uuwwww 10xxyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 0xDBFF 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 `decodeURIComponent` (

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. Escape sequences that could not have been introduced by `encodeURIComponent` 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 "#".

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 "#".

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 `unescapeURIComponentSet` be a String containing one instance of each code unit valid in `uriUnescaped`.
3. Return `Encode(componentString, unescapeURIComponentSet)`.

## 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 `ObjectCreateFromConstructor(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.

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 \(2^{\frac{1}{2}}\).

### 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
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.

#### 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`.
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).
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:

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, "", « »).

#### 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:


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 ? \( obj.\{\text{GetPrototypeOf}\}\)().

20.1.2.13  Object.is (\( value_1, value_2 \))

When the \texttt{is} function is called with arguments \( value_1 \) and \( value_2 \), the following steps are taken:

1. Return \texttt{SameValue(value_1, value_2)}.

20.1.2.14  Object.isExtensible (\( O \))

When the \texttt{isExtensible} function is called with argument \( O \), the following steps are taken:

1. If \texttt{Type(O)} is not Object, return \texttt{false}.
2. Return ? \texttt{IsExtensible(O)}.

20.1.2.15  Object.isFrozen (\( O \))

When the \texttt{isFrozen} function is called with argument \( O \), the following steps are taken:

1. If \texttt{Type(O)} is not Object, return \texttt{true}.
2. Return ? \texttt{TestIntegrityLevel(O, frozen)}.

20.1.2.16  Object.isSealed (\( O \))

When the \texttt{isSealed} function is called with argument \( O \), the following steps are taken:

1. If \texttt{Type(O)} is not Object, return \texttt{true}.
2. Return ? \texttt{TestIntegrityLevel(O, sealed)}.

20.1.2.17  Object.keys (\( O \))

When the \texttt{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 \texttt{preventExtensions} function is called, the following steps are taken:

1. If \texttt{Type(O)} is not Object, return \( O \).
2. Let \( status \) be ? \( O.\{\text{PreventExtensions}\}\)().
3. If \( status \) is \texttt{false}, throw a \texttt{TypeError} exception.
4. Return \( O \).

20.1.2.19  Object.prototype

The initial value of \texttt{Object.prototype} is the \texttt{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 ?\(\text{ToObject}(\text{this value})\).
3. Return ?\(\text{HasOwnProperty}(O, P)\).

**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 \textit{this} value is \textit{undefined} or \textit{null}.

### 20.1.3.3 \texttt{Object.prototype.isPrototypeOf(V)}

When the \texttt{isPrototypeOf} method is called with argument \(V\), the following steps are taken:

1. If \(\text{Type}(V)\) is not \texttt{Object}, return \textit{false}.
2. Let \(O\) be ?\(\text{ToObject}(\text{this value})\).
3. Repeat,
   a. Set \(V\) to \(V.\text{[[GetPrototypeOf]]}()\).
   b. If \(V\) is \texttt{null}, return \textit{false}.
   c. If \(\text{SameValue}(O, V)\) is \textit{true}, return \textit{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 \textit{this} value is \textit{undefined} or \textit{null}.

### 20.1.3.4 \texttt{Object.prototype.propertyIsEnumerable(V)}

When the \texttt{propertyIsEnumerable} method is called with argument \(V\), the following steps are taken:

1. Let \(P\) be ?\(\text{ToPropertyKey}(V)\).
2. Let \(O\) be ?\(\text{ToObject}(\text{this value})\).
3. Let \(desc\) be \(O.\text{[[GetOwnProperty]]}(P)\).
4. If \(desc\) is \texttt{undefined}, return \textit{false}.
5. Return \(desc.\text{[[Enumerable]]}\).

**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 \textit{this} value is \textit{undefined} or \textit{null}.

### 20.1.3.5 \texttt{Object.prototype.toLocaleString([reserved1 [, reserved2 ]])}

When the \texttt{toLocaleString} method is called, the following steps are taken:

1. Let \(O\) be the \textit{this} value.
2. Return ?\(\text{Invoke}(O, \text{"toString"})\).

The optional parameters to this function are not used but are intended to correspond to the parameter pattern used by ECMA-402 \texttt{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]].

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 \( \text{callerContext} \) be the second to top element of the execution context stack.
3. Let \( \text{callerRealm} \) be \( \text{callerContext}'s\ Realm.
4. Let \( \text{calleeRealm} \) be the current Realm Record.
5. Perform \(?\ \text{HostEnsureCanCompileStrings}(\text{callerRealm}, \text{calleeRealm})\).
6. If \( \text{newTarget} \) is undefined, set \( \text{newTarget} \) to constructor.
7. If \( \text{kind} \) is normal, then
   a. Let \( \text{goal} \) be the grammar symbol FunctionBody[\-Yield, \~Await].
   b. Let \( \text{parameterGoal} \) be the grammar symbol FormalParameters[\-Yield, \~Await].
   c. Let \( \text{fallbackProto} \) be "\%Function.prototype\%".
8. Else if \( \text{kind} \) is generator, then
   a. Let \( \text{goal} \) be the grammar symbol GeneratorBody.
   b. Let \( \text{parameterGoal} \) be the grammar symbol FormalParameters[+Yield, \~Await].
   c. Let \( \text{fallbackProto} \) be "\%GeneratorFunction.prototype\%".
9. Else if \( \text{kind} \) is async, then
   a. Let \( \text{goal} \) be the grammar symbol AsyncFunctionBody.
   b. Let \( \text{parameterGoal} \) be the grammar symbol FormalParameters[\-Yield, +Await].
   c. Let \( \text{fallbackProto} \) be "\%AsyncFunction.prototype\%".
10. Else,
    a. Assert: \( \text{kind} \) is asyncGenerator.
    b. Let \( \text{goal} \) be the grammar symbol AsyncGeneratorBody.
    c. Let \( \text{parameterGoal} \) be the grammar symbol FormalParameters[+Yield, +Await].
    d. Let \( \text{fallbackProto} \) be "\%AsyncGeneratorFunction.prototype\%".
11. Let \( \text{argCount} \) be the number of elements in \( \text{args} \).
12. Let \( P \) be the empty String.
13. If \( \text{argCount} = 0 \), let \( \text{bodyArg} \) be the empty String.
14. Else if \( \text{argCount} = 1 \), let \( \text{bodyArg} \) be \( \text{args}[0] \).
15. Else,
    a. Assert: \( \text{argCount} > 1 \).
    b. Let \( \text{firstArg} \) be \( \text{args}[0] \).
    c. Set \( P \) to \(?\ \text{ToString}(\text{firstArg})\).
    d. Let \( k \) be 1.
    e. Repeat, while \( k < \text{argCount} - 1 \),
       i. Let \( \text{nextArg} \) be \( \text{args}[k] \).
       ii. Let \( \text{nextArgString} \) be \(?\ \text{ToString}(\text{nextArg})\).
       iii. Set \( P \) to the string-concatenation of \( P \), "," (a comma), and \( \text{nextArgString} \).
       iv. Set \( k \) to \( k + 1 \).
    f. Let \( \text{bodyArg} \) be \( \text{args}[k] \).
16. Let \( \text{bodyString} \) be the string-concatenation of 0x000A (LINE FEED), \(?\ \text{ToString}(\text{bodyArg})\), and 0x000A (LINE FEED).
17. Let \( \text{prefix} \) be the prefix associated with \( \text{kind} \) in Table 50.
18. Let \( \text{sourceString} \) be the string-concatenation of \( \text{prefix} \), "\text{anonymous}(", \( P \), 0x000A (LINE FEED), ") \{", \( \text{bodyString} \), and ")".
19. Let \( \text{sourceText} \) be \(!\ \text{StringToCodePoints}(\text{sourceString})\).
20. Perform the following substeps in an implementation-defined order, possibly interleaving parsing and error detection:
    a. Let \( \text{parameters} \) be \( \text{ParseText}(!\ \text{StringToCodePoints}(P), \text{parameterGoal})\).
    b. If \( \text{parameters} \) 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.


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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>generator</td>
<td>&quot;function*&quot;</td>
</tr>
<tr>
<td>async</td>
<td>&quot;async function&quot;</td>
</tr>
<tr>
<td>asyncGenerator</td>
<td>&quot;async function*&quot;</td>
</tr>
</tbody>
</table>
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().
NOTE 1  The \textit{thisArg} value is passed without modification as the \textit{this} value. This is a change from Edition 3, where an \texttt{undefined} or \texttt{null} \textit{thisArg} is replaced with the \texttt{global object} and \texttt{ToObject} is applied to all other values and that result is passed as the \textit{this} value. Even though the \textit{thisArg} is passed without modification, non-strict functions still perform these transformations upon entry to the function.

NOTE 2  If \textit{func} is an arrow function or a \textit{bound function exotic object} then the \textit{thisArg} will be ignored by the function [[Call]] in step 6.

20.2.3.2 \texttt{Function.prototype.bind (thisArg, ...args)}

When the \texttt{bind} method is called with argument \textit{thisArg} and zero or more \textit{args}, it performs the following steps:

1. Let \textit{Target} be the \textit{this} value.
2. If \texttt{IsCallable(Target)} is \texttt{false}, throw a \texttt{TypeError} exception.
3. Let \textit{F} be \texttt{? BoundFunctionCreate(Target, thisArg, args)}.
4. Let \textit{L} be 0.
5. Let \textit{targetHasLength} be \texttt{? HasOwnProperty(Target, "length")}.
6. If \textit{targetHasLength} is \texttt{true}, then
   a. Let \textit{targetLen} be \texttt{? Get(Target, "length")}.
   b. If \texttt{Type(targetLen)} is Number, then
      i. If \textit{targetLen} is \texttt{+\infty}, set \textit{L} to \texttt{+\infty}.
      ii. Else if \textit{targetLen} is \texttt{-\infty}, set \textit{L} to 0.
      iii. Else,
          1. Let \textit{targetLenAsInt} be \texttt{! ToIntegerOrInfinity(targetLen)}.
          2. \texttt{Assert: targetLenAsInt} is finite.
          3. Let \textit{argCount} be the number of elements in \textit{args}.
          4. Set \textit{L} to \texttt{max(targetLenAsInt - argCount, 0)}.
   7. Perform \texttt{! SetFunctionLength(F, L)}.
8. Let \textit{targetName} be \texttt{? Get(Target, "name")}.
9. If \texttt{Type(targetName)} is not String, set \textit{targetName} to the empty String.
10. Perform \texttt{SetFunctionName(F, targetName, "bound")}.
11. Return \textit{F}.

NOTE 1  Function objects created using \texttt{Function.prototype.bind} are exotic objects. They also do not have a "prototype" property.

NOTE 2  If \textit{Target} is an arrow function or a \textit{bound function exotic object} then the \textit{thisArg} passed to this method will not be used by subsequent calls to \textit{F}.

20.2.3.3 \texttt{Function.prototype.call (thisArg, ...args)}

When the \texttt{call} method is called with argument \textit{thisArg} and zero or more \textit{args}, the following steps are taken:

1. Let \textit{func} be the \textit{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
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 `NativeFunctionAccessor opt PropertyName [~Yield, ~Await] opt` 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.

```javascript
NativeFunction : 
function NativeFunctionAccessor opt 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`.
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.

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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Key]]</td>
<td>A String</td>
<td>A string key used to globally identify a Symbol.</td>
</tr>
<tr>
<td>[[Symbol]]</td>
<td>A Symbol</td>
<td>A symbol that can be retrieved from any realm.</td>
</tr>
</tbody>
</table>

### 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`.

---

*498*
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`'s [[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`'s [[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` }.
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 \textbf{this} value.
2. If \text{Type}(O) is not Object, throw a \textbf{TypeError} exception.
3. Let \( name \) be \( \text{Get}(O, \text{"name"}) \).
4. If \( name \) is \textit{undefined}, set \( name \) to \"Error\"; otherwise set \( name \) to \( \text{ToString}(name) \).
5. Let \( msg \) be \( \text{Get}(O, \text{"message"}) \).
6. If \( msg \) is \textit{undefined}, set \( msg \) to the empty String; otherwise set \( msg \) to \( \text{ToString}(msg) \).
7. If \( name \) is the empty String, return \( msg \).
8. If \( msg \) is the empty String, return \( name \).
9. Return the \textit{string-concatenation} of \( name \), the code unit 0x003A (COLON), the code unit 0x0020 (SPACE), and \( msg \).

\textbf{20.5.4 Properties of Error Instances}

Error instances are ordinary objects that inherit properties from the \textit{Error prototype object} and have an [[ErrorData]] internal slot whose value is \textit{undefined}. The only specified uses of [[ErrorData]] is to identify Error, AggregateError, and \textit{NativeError} instances as Error objects within \textit{Object.prototype.toString}.

\textbf{20.5.5 Native Error Types Used in This Standard}

A new instance of one of the \textit{NativeError} objects below or of the AggregateError object is thrown when a runtime error is detected. All \textit{NativeError} objects share the same structure, as described in \textit{20.5.6}.

\textbf{20.5.5.1 EvalError}

The EvalError \textit{constructor} is \%EvalError\%.

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

\textbf{20.5.5.2 RangeError}

The RangeError \textit{constructor} is \%RangeError\%.

Indicates a value that is not in the set or range of allowable values.

\textbf{20.5.5.3 ReferenceError}

The ReferenceError \textit{constructor} is \%ReferenceError\%.

Indicate that an invalid reference has been detected.

\textbf{20.5.5.4 SyntaxError}

The SyntaxError \textit{constructor} is \%SyntaxError\%.

Indicates that a parsing error has occurred.

\textbf{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.

### 20.5.6 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.

20.5.7AggregateError 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
collectors 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 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 }.
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 `𝔽(ℝ(prim))`.
   c. Otherwise, let `n` be `prim`.
2. Else, let `n` be `+0𝔽`.
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 $2.220446049250313080847633361816 \times 10^{-16}$.

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`, or `-\infty`, 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 \( 9007199254740991 \) \( (2^{53} - 1) \).

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 \(-9007199254740991 \) \( (-(2^{53} - 1)) \).

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 \( 5 \times 10^{-324} \).

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 \(-\infty \).

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 \(+\infty\).

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 \(+0\)\(_F\).
- 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%`.

510
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 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 ≤ n < 10^{f+1} and for which n × 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 × 10^{e-f} is larger.
    b. Else,
       i. Let e, n, and f be integers such that f ≥ 0, 10^f ≤ n < 10^{f+1}, n × 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 ≠ 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 "x".
    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.
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 $f \geq 0$, $10^f \leq n < 10^{f+1}$, $n \times 10^e \cdot f$ is $x$, and $f$ is as small as possible. If there are multiple possibilities for $n$, choose the value of $n$ for which $n \times 10^e \cdot f$ 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(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 $m$. 
The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent Number values. For example,

\[(100000000000000128).toString()\] returns "100000000000000100", while \[(100000000000000128).toFixed(0)\] returns "100000000000000128".

### 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 `\(x\)`.
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 \(n \times 10^e \cdot p + 1 - x\) is as close to zero as possible. If there are two such sets of \(e\) and \(n\), pick the \(e\) and \(n\) for which \(n \times 10^e \cdot p + 1\) 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 \(e \geq p\), then
       i. Assert: \(e \neq 0\).
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 \( m \).
   b. 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 ] )

The optional \( \text{radix} \) should be an integral Number value in the inclusive range \( 2 \) to \( 36 \). If \( \text{radix} \) is \( \text{undefined} \) then \( 10 \) is used as the value of \( \text{radix} \).

The following steps are performed:

1. Let \( x \) be ? thisNumberValue(\( \text{this value} \)).
2. If \( \text{radix} \) is \( \text{undefined} \), let \( \text{radixMV} \) be 10.
3. Else, let \( \text{radixMV} \) be ? ToIntegerOrInfinity(\( \text{radix} \)).
4. If \( \text{radixMV} < 2 \) or \( \text{radixMV} > 36 \), throw a RangeError exception.
5. If \( \text{radixMV} = 10 \), return ! ToString(\( x \)).
6. Return the String representation of this Number value using the radix specified by \( \text{radixMV} \). Letters \( \text{a-z} \) 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 \( \text{this} \) 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 \( 1 \).

21.1.3.7 Number.prototype.valueOf ()

1. Return ? thisNumberValue(\( \text{this value} \)).
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 \( \mathbb{R}(bigint) \bmod 2^{bits} \).
4. If \( \text{mod} \geq 2^{\text{bits} - 1} \), return \( \mathbb{Z}(\text{mod} - 2^{\text{bits}}) \); otherwise, return \( \mathbb{Z}(\text{mod}) \).

21.2.2.2 BigInt.asUintN ( \( \text{bits}, \text{bigint} \) )

When the **BigInt.asUintN** function is called with two arguments \( \text{bits} \) and \( \text{bigint} \), the following steps are taken:

1. Set \( \text{bits} \) to \( \text{ToIndex}(\text{bits}) \).
2. Set \( \text{bigint} \) to \( \text{ToBigInt}(\text{bigint}) \).
3. Return the BigInt value that represents \( \mathbb{R}(\text{bigint}) \text{ modulo } 2^{\text{bits}} \).

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 \( \text{value} \). It performs the following steps when called:

1. If \( \text{Type(\text{value})} \) is BigInt, return \( \text{value} \).
2. If \( \text{Type(\text{value})} \) is Object and \( \text{value} \) has a [[BigIntData]] internal slot, then
   a. Assert: \( \text{Type(\text{value}.[[BigIntData]])} \) is BigInt.
   b. Return \( \text{value}.[[BigIntData]] \).
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 ( [ \( \text{reserved1} \), \( \text{reserved2} \) ] )

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 to 36. If radix is undefined then 10 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.LN10 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 ½, which is approximately 0.7071067811865476.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

NOTE The value of \( \text{Math}.\text{SQRT1_2} \) is approximately the reciprocal of the value of \( \text{Math}.\text{SQRT2} \).

21.3.1.8 \textbf{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 \textbf{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 \textbf{Function Properties of the Math Object}

NOTE The behaviour of the functions \texttt{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 \texttt{fdlibm}, the freely distributable mathematical library from Sun Microsystems (http://www.netlib.org/fdlibm).

21.3.2.1 \textbf{Math.abs ( x )}

Returns the absolute value of \( x \); the result has the same magnitude as \( x \) but has positive sign.

When the \texttt{Math.abs} method is called with argument \( x \), the following steps are taken:

1. Let \( n \) be ? \text{ToNumber}(x).
2. If \( n \) is \texttt{NaN}, return \texttt{NaN}.
3. If \( n \) is \(-0_\text{F} \), return \(+0_\text{F} \).
4. If \( n \) is \( -\infty_\text{F} \), return \( +\infty_\text{F} \).
5. If \( n < +0_\text{F} \), return \( -n \).
6. Return \( n \).

21.3.2.2 \textbf{Math.acos ( x )}
Returns the inverse cosine of $x$. The result is expressed in radians and ranges from $+0_F$ to $\pi$, inclusive.

When the **Math.acos** method is called with argument $x$, the following steps are taken:

1. Let $n$ be $\text{ToNumber}(x)$.
2. If $n$ is NaN, $n > 1_F$, or $n < -1_F$, return NaN.
3. If $n$ is $1_F$, return $+0_F$.
4. Return an implementation-approximated value representing the result of the inverse cosine of $R(n)$.

### 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 $\text{ToNumber}(x)$.
2. If $n$ is NaN or $n$ is $+\infty_F$, return $n$.
3. If $n$ is $1_F$, return $+0_F$.
4. If $n < 1_F$, return NaN.
5. Return an implementation-approximated value representing the result of the inverse hyperbolic cosine of $R(n)$.

### 21.3.2.4 Math.asin ($x$)

Returns the inverse sine of $x$. The result is expressed in radians and ranges from $-\pi / 2$ to $\pi / 2$, inclusive.

When the **Math.asin** method is called with argument $x$, the following steps are taken:

1. Let $n$ be $\text{ToNumber}(x)$.
2. If $n$ is NaN or $n$ is $+0_F$, or $n$ is $-0_F$, return $n$.
3. If $n > 1_F$ or $n < -1_F$, return NaN.
4. Return an implementation-approximated value representing the result of the inverse sine of $R(n)$.

### 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 $\text{ToNumber}(x)$.
2. If $n$ is NaN, $n$ is $+0_F$, $n$ is $-0_F$, $n$ is $+\infty_F$, or $n$ is $-\infty_F$, return $n$.
3. Return an implementation-approximated value representing the result of the inverse hyperbolic sine of $R(n)$.

### 21.3.2.6 Math.atan ($x$)

Returns the inverse tangent of $x$. The result is expressed in radians and ranges from $-\pi / 2$ to $\pi / 2$, inclusive.

When the **Math.atan** method is called with argument $x$, the following steps are taken:

1. Let $n$ be $\text{ToNumber}(x)$.
2. If $n$ is NaN, $n$ is $+0_F$, $n$ is $-0_F$, return $n$. 
3. If \( n \) is \(+\infty\), return an implementation-approximated value representing \( \pi / 2 \).
4. If \( n \) is \(-\infty\), 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 \(-\pi / 4 \).
   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 \( \pi \).
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 $-\infty$, return an implementation-approximated value representing $\pi$.
c. If $nx$ is $+0_F$ or $nx$ is $-0_F$, return an implementation-approximated value representing $\pi / 2$.

10. If $ny < +0_F$, then
   a. If $nx$ is $+\infty_F$, return $-0_F$.
   b. If $nx$ is $-\infty_F$, return an implementation-approximated value representing $-\pi$.
   c. If $nx$ is $+0_F$ or $nx$ is $-0_F$, return an implementation-approximated value representing $-\pi / 2$.

11. Assert: $nx$ is finite and is neither $+0_F$ nor $-0_F$.

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 $+0_F$, $n$ is $-0_F$, $n$ is $+\infty_F$, or $n$ is $-\infty_F$, 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 $+0_F$, $n$ is $-0_F$, $n$ is $+\infty_F$, or $n$ is $-\infty_F$, return $n$.
3. If $n < +0_F$ and $n > -1_F$, return $-0_F$.
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 $-\text{Math.floor}(-x)$.

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 $\mathbb{F}(p)$.

NOTE If $n$ is $+0_F$ or $n$ is $-0_F$, this method returns $32_F$. If the most significant bit of the 32-bit binary encoding of $n$ is 1, this method returns $+0_F$. 
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 \(\text{ToNumber}(x)\).
2. If \(n\) is `NaN`, \(n\) is \(+\infty\) or \(n\) is \(-\infty\), return \(n\).
3. If \(n\) is \(+\infty\) or \(n\) is \(-\infty\), 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 \(\text{ToNumber}(x)\).
2. If \(n\) is `NaN`, \(n\) is \(+\infty\) or \(n\) is \(-\infty\), return \(n\).
3. If \(n\) is \(+0\) or \(n\) is \(-0\), return \(1\).
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 \((\text{Math.exp}(x) + \text{Math.exp}(-x)) / 2\).

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 \(\text{ToNumber}(x)\).
2. If \(n\) is `NaN` or \(n\) is \(+\infty\), return \(n\).
3. If \(n\) is \(+0\) or \(n\) is \(-0\), return \(1\).
4. If \(n\) is \(-\infty\), return \(-1\).
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 \(\text{ToNumber}(x)\).
2. If \(n\) is `NaN`, \(n\) is \(+0\), \(n\) is \(-0\) or \(n\) is \(+\infty\), return \(n\).
3. If \(n\) is \(-\infty\), return \(-1\).
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 +∞) 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\(_F\), n is -0\(_F\), n is +\(\infty\)\(_F\), or n is -\(\infty\)\(_F\), return n.
3. If n < 1\(_F\) and n > +0\(_F\), return +0\(_F\).
4. If n is an integral Number, return n.
5. Return the greatest (closest to +∞) 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\(_F\), -0\(_F\), +\(\infty\)\(_F\), or -\(\infty\)\(_F\), 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 +\(\infty\)\(_F\), return number.
   b. If number is -\(\infty\)\(_F\), return +\(\infty\)\(_F\).
   c. If number is neither +0\(_F\) nor -0\(_F\), set onlyZero to false.
5. If onlyZero is true, return +0\(_F\).
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\(_F\).
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 ℝ(? ToUint32(x)).
2. Let b be ℝ(? ToUint32(y)).
3. Let product be (a × b) modulo 2^{32}.

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 +∞, return n.
3. If n is 1 return +0.
4. If n is +0 or n is -0, return -∞.
5. If n < +0, return NaN.
6. Return an implementation-approximated value representing the result of the natural logarithm of 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, n is -0, or n is +∞, return n.
3. If n is -1 return -∞.
4. If n < -1, return NaN.
5. Return an implementation-approximated value representing the result of the natural logarithm of 1 + 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 +∞, return n.
3. If n is 1 return +0.
4. If $n$ is $+0_F$ or $n$ is $-0_F$, return $-\infty_F$.
5. If $n < +0_F$, 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 $+\infty_F$, return $n$.
3. If $n$ is $1_F$, return $+0_F$.
4. If $n$ is $+0_F$ or $n$ is $-0_F$, return $-\infty_F$.
5. If $n < +0_F$, 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 $-\infty_F$.
4. For each element number of coerced, do
   a. If number is NaN, return NaN.
   b. If number is $+0_F$ and highest is $-0_F$, set highest to $+0_F$.
   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 $+0_F$ is considered to be larger than $-0_F$.

The "length" property of the max method is 2.$F$.

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 \( \text{ToNumber}(\text{arg}) \).
   b. Append \( n \) to coerced.
3. Let lowest be \( +\infty \).
4. For each element number of coerced, do
   a. If number is NaN, return NaN.
   b. If number is \( -0 \) and lowest is \( +0 \), set lowest to \( -0 \).
   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 \( +0 \) is considered to be larger than \( -0 \).

The "length" property of the \texttt{min} method is \( 2 \).

21.3.2.26 \texttt{Math.pow (base, exponent)}

When the \texttt{Math.pow} method is called with arguments \textit{base} and \textit{exponent}, the following steps are taken:

1. Set \textit{base} to \( \text{ToNumber}(	ext{base}) \).
2. Set \textit{exponent} to \( \text{ToNumber}(	ext{exponent}) \).
3. Return \( \text{Number::exponentiate}(	ext{base}, \text{exponent}) \).

21.3.2.27 \texttt{Math.random ()}

Returns a \textit{Number value} with positive sign, greater than or equal to \( +0 \) but strictly less than \( 1 \), 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 \texttt{Math.random} function created for distinct realms must produce a distinct sequence of values from successive calls.

21.3.2.28 \texttt{Math.round (x)}

Returns the \textit{Number value} that is closest to \( x \) and is integral. If two integral Numbers are equally close to \( x \), then the result is the \textit{Number value} that is closer to \( +\infty \). If \( x \) is already integral, the result is \( x \).

When the \texttt{Math.round} method is called with argument \( x \), the following steps are taken:

1. Let \( n \) be \( \text{ToNumber}(x) \).
2. If \( n \) is NaN, \( +\infty \), \( -\infty \), or an integral Number, return \( n \).
3. If \( n < 0.5 \) and \( n > -0 \), return \( +0 \).
4. If \( n < +0 \) and \( n \geq -0.5 \), return \( -0 \).
5. Return the integral Number closest to \( n \), preferring the Number closer to \( +\infty \) in the case of a tie.

NOTE 1 \texttt{Math.round(3.5)} returns 4, but \texttt{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\( _F \) but greater than or equal to -0.5\( _F \), Math.round(x) returns -0\( _F \) but Math.floor(x + 0.5) returns +0\( _F \).
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\( _F \), or n is -0\( _F \), return n.
3. If n < +0\( _F \), 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\( _F \), or n is -0\( _F \), return n.
3. If n is +\( +\infty _F \) or n is -\( -\infty _F \), 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\( _F \), n is -0\( _F \), n is +\( +\infty _F \), or n is -\( -\infty _F \), 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 \( \text{NaN} \), \( n \) is +0, or \( n \) is +\( \infty \), return \( n \).
3. If \( n < +0 \), return \( \text{NaN} \).
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 \texttt{Math.tan} method is called with argument \( x \), the following steps are taken:

1. Let \( n \) be \( \text{ToNumber}(x) \).
2. If \( n \) is \( \text{NaN} \), \( n \) is +0, or \( n \) is -0, return \( n \).
3. If \( n \) is +\( \infty \) or \( n \) is -\( \infty \), return \( \text{NaN} \).
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 \texttt{Math.tanh} method is called with argument \( x \), the following steps are taken:

1. Let \( n \) be \( \text{ToNumber}(x) \).
2. If \( n \) is \( \text{NaN} \), \( n \) is +0, or \( n \) is -0, return \( n \).
3. If \( n \) is +\( \infty \), return 1.
4. If \( n \) is -\( \infty \), return -1.
5. Return an implementation-approximated value representing the result of the hyperbolic tangent of \( \mathbb{R}(n) \).

#### NOTE

The value of \texttt{Math.tanh(x)} is the same as the value of

\[
\frac{\text{Math.exp}(x) - \text{Math.exp}(-x)}{\text{Math.exp}(x) + \text{Math.exp}(-x)}.
\]

### 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 \texttt{Math.trunc} method is called with argument \( x \), the following steps are taken:

1. Let \( n \) be \( \text{ToNumber}(x) \).
2. If \( n \) is \( \text{NaN} \), \( n \) is +0, \( n \) is -0, \( n \) is +\( \infty \), or \( n \) is -\( \infty \), return \( n \).
3. If \( n < 1 \) and \( n > +0 \), return +0.
4. If \( n < +0 \) and \( n > -1 \), return -0.
5. Return the integral Number nearest \( n \) in the direction of +0.

### 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 $\pm 0$.  

```plaintext
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

$$\text{Day}(t) = \lfloor \text{floor}(t / \text{msPerDay}) \rfloor$$

where the number of milliseconds per day is

$$\text{msPerDay} = 86400000_\mathbb{F}$$

The remainder is called the time within the day:

$$\text{TimeWithinDay}(t) = \lfloor t \mod \text{msPerDay} \rfloor$$

### 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

$$\text{DaysInYear}(y) = 365\frac{\mu}{\nu} \text{ if } (\mathbb{R}(y) \text{ modulo } 4) \neq 0$$
$$= 366\frac{\mu}{\nu} \text{ if } (\mathbb{R}(y) \text{ modulo } 4) = 0 \text{ and } (\mathbb{R}(y) \text{ modulo } 100) \neq 0$$
$$= 365\frac{\mu}{\nu} \text{ if } (\mathbb{R}(y) \text{ modulo } 100) = 0 \text{ and } (\mathbb{R}(y) \text{ modulo } 400) \neq 0$$
$$= 366\frac{\mu}{\nu} \text{ if } (\mathbb{R}(y) \text{ modulo } 400) = 0$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year $y$ is given by:

$$\text{DayFromYear}(y) = 365 \times (\mathbb{R}(y) - 1970) + \text{floor}(\mathbb{R}(y) - 1969) / 4 - \text{floor}(\mathbb{R}(y) - 1901) / 100 + \text{floor}(\mathbb{R}(y) - 1601) / 400)$$

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 $+0_{\mathbb{F}}$:

$$\text{InLeapYear}(t) = +0_{\mathbb{F}} \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 365_{\mathbb{F}}$$
$$= 1_{\mathbb{F}} \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 366_{\mathbb{F}}$$

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 $\text{MonthFromTime}(t)$ from a time value $t$ to a month number is defined by:

$$\text{MonthFromTime}(t) = +0_{\mathbb{F}} \text{ if } +0_{\mathbb{F}} \leq \text{DayWithinYear}(t) < 31_{\mathbb{F}}$$
$$= 1_{\mathbb{F}} \text{ if } 31_{\mathbb{F}} \leq \text{DayWithinYear}(t) < 59_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 2_{\mathbb{F}} \text{ if } 59_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 3_{\mathbb{F}} \text{ if } 90_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 4_{\mathbb{F}} \text{ if } 120_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 5_{\mathbb{F}} \text{ if } 151_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 6_{\mathbb{F}} \text{ if } 181_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 7_{\mathbb{F}} \text{ if } 212_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 8_{\mathbb{F}} \text{ if } 243_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 9_{\mathbb{F}} \text{ if } 273_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 10_{\mathbb{F}} \text{ if } 304_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334_{\mathbb{F}} + \text{InLeapYear}(t)$$
$$= 11_{\mathbb{F}} \text{ if } 334_{\mathbb{F}} + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365_{\mathbb{F}} + \text{InLeapYear}(t)$$

where
DayWithinYear(t) = Day(t) - DayFromYear(YearFromTime(t))

A month value of \(+0\) specifies January; \(1\) specifies February; \(2\) specifies March; \(3\) specifies April; \(4\) specifies May; \(5\) specifies June; \(6\) specifies July; \(7\) specifies August; \(8\) specifies September; \(9\) specifies October; \(10\) specifies November; and \(11\) specifies December. Note that MonthFromTime\(+0\) = \(+0\), corresponding to Thursday, 1 January 1970.

21.4.1.5 Date Number

A date number is identified by an integral Number in the range \(1\) through \(31\), inclusive. The mapping DateFromTime(t) from a time value t to a date number is defined by:

- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) + 1\) if MonthFromTime(t) = \(+0\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 30\) if MonthFromTime(t) = \(1\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 58 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(2\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 89 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(3\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 119 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(4\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 150 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(5\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 180 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(6\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 211 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(7\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 242 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(8\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 272 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(9\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 303 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(10\)
- \(\text{DateFromTime}(t) = \text{DayWithinYear}(t) - 333 - \text{InLeapYear}(t)\) if MonthFromTime(t) = \(11\)

21.4.1.6 Week Day

The weekday for a particular time value t is defined as

\[
\text{WeekDay}(t) = \mathbb{R}(\text{Day}(t) + 4) \mod 7
\]

A weekday value of \(+0\) specifies Sunday; \(1\) specifies Monday; \(2\) specifies Tuesday; \(3\) specifies Wednesday; \(4\) specifies Thursday; \(5\) specifies Friday; and \(6\) specifies Saturday. Note that WeekDay\(+0\) = \(4\), corresponding to Thursday, 1 January 1970.

21.4.1.7 LocalTZA( t, isUTC )

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, LocalTZA( tUTC, true ) should return the offset of the local time zone from UTC measured in milliseconds at time represented by time value tUTC. When the result is added to tUTC, it should yield the corresponding Number tlocal.

When isUTC is false, LocalTZA( tlocal, 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 +08.

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.
\[
\text{LocalTZA(TimeClip(MakeDate(MakeDay(2017, 10, 5), MakeTime(1, 30, 0, 0))), false) is } -4 \times \text{msPerHour}. 
\]

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).
\[
\text{LocalTZA(TimeClip(MakeDate(MakeDay(2017, 2, 12), MakeTime(2, 30, 0, 0))), false) is } -5 \times \text{msPerHour}. 
\]

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.).

\[
\text{LocalTime(UTC(} t_{local} \text{)) is not necessarily always equal to } t_{local}. \text{ Correspondingly, UTC(LocalTime(} t_{UTC} \text{)) is not necessarily always equal to } t_{UTC}. 
\]

**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 $t_{\text{UTC}}$. Correspondingly, $\text{LocalTime}(\text{UTC}(t_{\text{local}}))$ is not necessarily always equal to $t_{\text{local}}$.

21.4.1.10 Hours, Minutes, Second, and Milliseconds

The following abstract operations are useful in decomposing time values:

- $\text{HourFromTime}(t) = \text{floor}(\mathbb{R}(t / \text{msPerHour})) \mod \text{HoursPerDay}$
- $\text{MinFromTime}(t) = \text{floor}(\mathbb{R}(t / \text{msPerMinute})) \mod \text{MinutesPerHour}$
- $\text{SecFromTime}(t) = \text{floor}(\mathbb{R}(t / \text{msPerSecond})) \mod \text{SecondsPerMinute}$
- $\text{msFromTime}(t) = \mathbb{R}(t) \mod \text{msPerSecond}$

where

- $\text{HoursPerDay} = 24$
- $\text{MinutesPerHour} = 60$
- $\text{SecondsPerMinute} = 60$
- $\text{msPerSecond} = 1000$
- $\text{msPerMinute} = 60000 = \text{msPerSecond} \times \text{SecondsPerMinute}$
- $\text{msPerHour} = 3600000 = \text{msPerMinute} \times \text{MinutesPerHour}$

21.4.1.11 MakeTime ($\text{hour}$, $\text{min}$, $\text{sec}$, $\text{ms}$)

The abstract operation MakeTime takes arguments $\text{hour}$ (a Number), $\text{min}$ (a Number), $\text{sec}$ (a Number), and $\text{ms}$ (a Number). It calculates a number of milliseconds. It performs the following steps when called:

1. If $\text{hour}$ is not finite or $\text{min}$ is not finite or $\text{sec}$ is not finite or $\text{ms}$ is not finite, return NaN.
2. Let $h$ be $\text{ToIntegerOrInfinity}(\text{hour})$.
3. Let $m$ be $\text{ToIntegerOrInfinity}(\text{min})$.
4. Let $s$ be $\text{ToIntegerOrInfinity}(\text{sec})$.
5. Let $\text{milli}$ be $\text{ToIntegerOrInfinity}(\text{ms})$.
6. Let $t$ be $(h \times \text{msPerHour} + m \times \text{msPerMinute} + s \times \text{msPerSecond}) + \text{milli}$, performing the arithmetic according to IEEE 754-2019 rules (that is, as if using the ECMAScript operators $*$ and $+$).
7. Return $t$.

21.4.1.12 MakeDay ($\text{year}$, $\text{month}$, $\text{date}$)

The abstract operation MakeDay takes arguments $\text{year}$ (a Number), $\text{month}$ (a Number), and $\text{date}$ (a Number). It calculates a number of days. It performs the following steps when called:

1. If $\text{year}$ is not finite or $\text{month}$ is not finite or $\text{date}$ is not finite, return NaN.
2. Let $y$ be $\text{ToIntegerOrInfinity}(\text{year})$.
3. Let $m$ be $\text{ToIntegerOrInfinity}(\text{month})$.
4. Let $\text{dt}$ be $\text{ToIntegerOrInfinity}(\text{date})$.
5. Let $\text{ym} = y + \text{floor}(\mathbb{R}(m / 12))$.
6. If $\text{ym}$ is not finite, return NaN.
7. Let \( mn \) be \( \mathbb{F}(m) \mod 12 \).
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 \( 1 \); but if this is not possible (because some argument is out of range), return \( \text{NaN} \).
9. Return \( \text{Day}(t) + dt - 1 \).

### 21.4.1.13 MakeDate ( \( day, time \) )

The abstract operation \( \text{MakeDate} \) 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 \( \text{NaN} \).
2. Let \( tv \) be \( day \times \text{msPerDay} + time \).
3. If \( tv \) is not finite, return \( \text{NaN} \).
4. Return \( tv \).

### 21.4.1.14 TimeClip ( \( time \) )

The abstract operation \( \text{TimeClip} \) 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 \( \text{NaN} \).
2. If \( |\mathbb{R}(time)| > 8.64 \times 10^{15} \), return \( \text{NaN} \).
3. Return \( \overline{!} \overline{!} \overline{\text{ToIntegerOrInfinity}}(time) \).

### 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 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:
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.

### 21.4.15.1 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.
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.

21.4.2 The Date Constructor

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 1E.
   e. If numberOfArgs > 3, let h be ? ToNumber(values[3]); else let h be +0E.
   f. If numberOfArgs > 4, let min be ? ToNumber(values[4]); else let min be +0E.
   g. If numberOfArgs > 5, let s be ? ToNumber(values[5]); else let s be +0E.
   h. If numberOfArgs > 6, let milli be ? ToNumber(values[6]); else let milli be +0E.
   i. If y is NaN, let yr be NaN.
   j. Else,
i. Let \( y_i \) be \( \text{ToIntegerOrInfinity}(y) \).
ii. If \( 0 \leq y_i \leq 99 \), let \( y_r \) be \( 1900 + f(y_i) \); otherwise, let \( y_r \) be \( y \).

k. Let \( \text{finalDate} \) be \( \text{MakeDate(MakeDay}(y_r, m, dt), \text{MakeTime}(h, \text{min}, s, \text{milli})) \).

1. Let \( dv \) be \( \text{TimeClip(UTC(finalDate))} \).

6. Let \( O \) be \( \text{? OrdinaryCreateFromConstructor(NewTarget, "%Date.prototype", « [[DateValue]] »)} \).
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:

\[
\begin{align*}
\text{\text{\text{x.valueOf()}}} & \\
\text{\text{Date.parse(x.toString())}} & \\
\text{\text{Date.parse(x.toUTCString())}} & \\
\text{\text{Date.parse(x.toISOString())}} & \\
\text{\text{Date.parse(x.toLocaleString())}} & 
\end{align*}
\]

However, the expression

\[
\text{\text{Date.parse(x.toLocaleDateString())}}
\]

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 (21.4.15) 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 +0E.
3. If `date` is present, let \( dt \) be ? `ToNumber(date)`; else let \( dt \) be 1E.
4. If `hours` is present, let \( h \) be ? `ToNumber(hours)`; else let \( h \) be +0E.
5. If `minutes` is present, let \( min \) be ? `ToNumber(minutes)`; else let \( min \) be +0E.
6. If `seconds` is present, let \( s \) be ? `ToNumber(seconds)`; else let \( s \) be +0E.
7. If `ms` is present, let `milli` be ? `ToNumber(ms)`; else let `milli` be +0E.
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 \( 1900E + F(yi) \); 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 7E.

**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 \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{msFromTime} (\text{LocalTime}(t)) \).

21.4.4.7  \text{Date.prototype.getMinutes} ( )

The following steps are performed:

1. Let \( t \) be ? \text{thisTimeValue} (\text{this} \ \text{value}).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MinFromTime} (\text{LocalTime}(t)) \).

21.4.4.8  \text{Date.prototype.getMonth} ( )

The following steps are performed:

1. Let \( t \) be ? \text{thisTimeValue} (\text{this} \ \text{value}).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MonthFromTime} (\text{LocalTime}(t)) \).

21.4.4.9  \text{Date.prototype.getSeconds} ( )

The following steps are performed:

1. Let \( t \) be ? \text{thisTimeValue} (\text{this} \ \text{value}).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{SecFromTime} (\text{LocalTime}(t)) \).

21.4.4.10  \text{Date.prototype.getTime} ( )

The following steps are performed:

1. Return ? \text{thisTimeValue} (\text{this} \ \text{value}).

21.4.4.11  \text{Date.prototype.getTimezoneOffset} ( )

The following steps are performed:

1. Let \( t \) be ? \text{thisTimeValue} (\text{this} \ \text{value}).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( (t - \text{LocalTime}(t)) \) / \( \text{msPerMinute} \).

21.4.4.12  \text{Date.prototype.getUTCDate} ( )

The following steps are performed:

1. Let \( t \) be ? \text{thisTimeValue} (\text{this} \ \text{value}).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{DateFromTime}(t) \).

21.4.4.13  \text{Date.prototype.getUTCDay} ( )
The following steps are performed:

1. Let $t$ be $\text{thisTimeValue}(\text{this value})$.
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 $\text{thisTimeValue}(\text{this value})$.
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 $\text{thisTimeValue}(\text{this value})$.
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 $\text{thisTimeValue}(\text{this value})$.
2. If $t$ is NaN, return NaN.
3. Return msFromTime($t$).

21.4.4.17 Date.prototype.getUTCMilliseconds ()

The following steps are performed:

1. Let $t$ be $\text{thisTimeValue}(\text{this value})$.
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 $\text{thisTimeValue}(\text{this value})$.
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 $\text{thisTimeValue}(\text{this value})$. 
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 +0; 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 3.

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.

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.

The "length" property of the setMinutes method is 3.

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().

21.4.4.24 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.

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 $2^\mathbb{F}$.

**NOTE**  
If `date` is not present, this method behaves as if `date` was present with the value `getDateTime()`.

### 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 $2^\mathbb{F}$.

**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 $+0_\mathbb{F}$.  

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.

The "length" property of the setUTCFullYear method is 3.

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.

The "length" property of the setUTCHours method is 4.

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 ] ] )

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().

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().
The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. Let \( m \) be \( \text{ToNumber}(\text{mini}) \).
3. If \( \text{sec} \) is not present, let \( s \) be \( \text{SecFromTime}(t) \).
4. Else,
   a. Let \( s \) be \( \text{ToNumber}(\text{sec}) \).
5. If \( \text{ms} \) is not present, let \( \text{milli} \) be \( \text{msFromTime}(t) \).
6. Else,
   a. Let \( \text{milli} \) be \( \text{ToNumber}(\text{ms}) \).
7. Let \( \text{date} \) be \( \text{MakeDate}(\text{Day}(t), \text{MakeTime}(\text{HourFromTime}(t), m, s, \text{milli})) \).
8. Let \( v \) be \( \text{TimeClip}(\text{date}) \).
9. Set the \([\text{[DateValue]}]\) internal slot of this Date object to \( v \).
10. Return \( v \).

The "length" property of the \textit{setUTCMinutes} method is \( 3 \). 

\begin{note}
If \( \text{sec} \) is not present, this method behaves as if \( \text{sec} \) was present with the value \( \text{getUTCSecs()}. \) If \( \text{ms} \) is not present, it function behaves as if \( \text{ms} \) was present with the value return by \( \text{getUTCMilliseconds()} \).
\end{note}

21.4.4.33 \textit{Date.prototype.setUTCMonth} ( \textit{month} [, \textit{date} ] )

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. Let \( m \) be \( \text{ToNumber}(\text{month}) \).
3. If \( \text{date} \) is not present, let \( dt \) be \( \text{DateFromTime}(t) \).
4. Else,
   a. Let \( dt \) be \( \text{ToNumber}(\text{date}) \).
5. Let \( \text{newDate} \) be \( \text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), m, dt), \text{TimeWithinDay}(t)) \).
6. Let \( v \) be \( \text{TimeClip}(\text{newDate}) \).
7. Set the \([\text{[DateValue]}]\) internal slot of this Date object to \( v \).
8. Return \( v \).

The "length" property of the \textit{setUTCMonth} method is \( 2 \). 

\begin{note}
If \( \text{date} \) is not present, this method behaves as if \( \text{date} \) was present with the value \( \text{getUTCDate()} \).
\end{note}

21.4.4.34 \textit{Date.prototype.setUTCSecs} ( \textit{sec} [, \textit{ms} ] )

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. Let \( s \) be \( \text{ToNumber}(\text{sec}) \).
3. If \( \text{ms} \) is not present, let \( \text{milli} \) be \( \text{msFromTime}(t) \).
4. Else,
a. Let \( \text{milli} \) be \( \text{ToNumber}(ms) \).
5. Let \( \text{date} \) be \( \text{MakeDate(Day}(t), \text{MakeTime(HourFromTime}(t), \text{MinFromTime}(t), s, \text{milli})) \).
6. Let \( \nu \) be \( \text{TimeClip(date)} \).
7. Set the [[DateValue]] internal slot of this Date object to \( \nu \).
8. Return \( \nu \).

The "length" property of the \texttt{setUTCSeconds} method is \( 2^6 \).

\begin{note}
If \( ms \) is not present, this method behaves as if \( ms \) was present with the value \texttt{getUTCMilliseconds()}.
\end{note}

21.4.4.35 \texttt{Date.prototype.toDateString()} 

The following steps are performed:

1. Let \( O \) be this Date object.
2. Let \( tv \) be \( \text{thisTimeValue}(O) \).
3. If \( tv \) is \( \text{NaN} \), return "Invalid Date".
4. Let \( t \) be \( \text{LocalTime}(tv) \).
5. Return \( \text{DateString}(t) \).

21.4.4.36 \texttt{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 \texttt{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 \texttt{Date.prototype.toJSON(key)}

This function provides a String representation of a Date object for use by \texttt{JSON.stringify(25.5.2)}.

When the \texttt{toJSON} method is called with argument \( key \), the following steps are taken:

1. Let \( O \) be \( \text{ToObject(this value)} \).
2. Let \( tv \) be \( \text{ToPrimitive}(O, \text{number}) \).
3. If Type(\( tv \)) is Number and \( tv \) is not finite, return \texttt{null}.
4. Return \( \text{Invoke}(O, \text{"toISOString"}) \).

\begin{note1}
The argument is ignored.
\end{note1}

\begin{note2}
The \texttt{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 \texttt{toISOString} method.
\end{note2}

21.4.4.38 \texttt{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.

21.4.39  Date.prototype.toLocaleString ([ 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 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.

21.4.40  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.

21.4.41  Date.prototype.toString ()

The following steps are performed:

1. Let tv be ? thisTimeValue(this value).
2. Return ToDateString(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 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 \( yv \geq +0_F \), let \( yearSign \) be the empty String; otherwise, let \( yearSign \) be "-".
8. Let \( year \) be the String representation of abs(\( yv \)), formatted as a decimal number.
9. Let \( paddedYear \) be \( ! \) StringPad(\( year \), 4_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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0_F</td>
<td>&quot;Sun&quot;</td>
</tr>
<tr>
<td>1_F</td>
<td>&quot;Mon&quot;</td>
</tr>
<tr>
<td>2_F</td>
<td>&quot;Tue&quot;</td>
</tr>
<tr>
<td>3_F</td>
<td>&quot;Wed&quot;</td>
</tr>
<tr>
<td>4_F</td>
<td>&quot;Thu&quot;</td>
</tr>
<tr>
<td>5_F</td>
<td>&quot;Fri&quot;</td>
</tr>
<tr>
<td>6_F</td>
<td>&quot;Sat&quot;</td>
</tr>
</tbody>
</table>
Table 53: Names of months of the year

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0xFF</td>
<td>&quot;Jan&quot;</td>
</tr>
<tr>
<td>1xFF</td>
<td>&quot;Feb&quot;</td>
</tr>
<tr>
<td>2xFF</td>
<td>&quot;Mar&quot;</td>
</tr>
<tr>
<td>3xFF</td>
<td>&quot;Apr&quot;</td>
</tr>
<tr>
<td>4xFF</td>
<td>&quot;May&quot;</td>
</tr>
<tr>
<td>5xFF</td>
<td>&quot;Jun&quot;</td>
</tr>
<tr>
<td>6xFF</td>
<td>&quot;Jul&quot;</td>
</tr>
<tr>
<td>7xFF</td>
<td>&quot;Aug&quot;</td>
</tr>
<tr>
<td>8xFF</td>
<td>&quot;Sep&quot;</td>
</tr>
<tr>
<td>9xFF</td>
<td>&quot;Oct&quot;</td>
</tr>
<tr>
<td>10xFF</td>
<td>&quot;Nov&quot;</td>
</tr>
<tr>
<td>11xFF</td>
<td>&quot;Dec&quot;</td>
</tr>
</tbody>
</table>

21.4.41.3 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 ≥ +0xFF, 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.

21.4.41.4 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 $yv \geq +0F$, let yearSign be the empty String; otherwise, let yearSign be "+".
9. Let year be the String representation of abs$(\Re(yv))$, 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 \( \text{ToString}(value) \).

3. If NewTarget is \text{undefined}, return \( s \).

4. Return \! \text{StringCreate}(s, \text{? GetPrototypeFromConstructor}(\text{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 \text{String.fromCharCode} function may be called with any number of arguments which form the rest parameter \text{codeUnits}. The following steps are taken:

1. Let \( \text{length} \) be the number of elements in \text{codeUnits}.
2. Let \( \text{elements} \) be a new empty \text{List}.
3. For each element \( \text{next} \) of \text{codeUnits}, do
   a. Let \( \text{nextCU} \) be \( \mathbb{R}(\text{? ToUint16}(\text{next})) \).
   b. Append \( \text{nextCU} \) to the end of \( \text{elements} \).
4. Return the String value whose code units are the elements in the \text{List elements}. If \text{codeUnits} is empty, the empty String is returned.

The "\text{length}" property of the \text{fromCharCode} function is 1\(_\mathbb{F}\).

#### 22.1.2.2 String.fromCodePoint ( ...codePoints )

The \text{String.fromCodePoint} function may be called with any number of arguments which form the rest parameter \text{codePoints}. The following steps are taken:

1. Let \( \text{result} \) be the empty String.
2. For each element \( \text{next} \) of \text{codePoints}, do
   a. Let \( \text{nextCP} \) be \( \text{? ToNumber}(\text{next}) \).
   b. If \( \text{IsIntegralNumber}(\text{nextCP}) \) is false, throw a \text{RangeError} exception.
   c. If \( \mathbb{R}(\text{nextCP}) < 0 \) or \( \mathbb{R}(\text{nextCP}) > 0x10FFFF \), throw a \text{RangeError} exception.
   d. Set \( \text{result} \) to the string-concatenation of \( \text{result} \) and \( \text{! UTF16EncodeCodePoint}(\mathbb{R}(\text{nextCP})) \).
3. Assert: If \text{codePoints} is empty, then \( \text{result} \) is the empty String.
4. Return \( \text{result} \).

The "\text{length}" property of the \text{fromCodePoint} function is 1\(_\mathbb{F}\).

#### 22.1.2.3 String.prototype

The initial value of \text{String.prototype} is the \text{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 +0𝔽 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 \( position \lt 0 \) or \( position \geq size \), 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 \( position \lt 0 \) or \( position \geq size \), 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)

255
NOTE 1

Returns a non-negative integral Number less than or equal to \(0x10FFFF\) 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 \texttt{undefined}. If a valid UTF-16 surrogate pair does not begin at \(pos\), the result is the code unit at \(pos\).

When the \texttt{codePointAt} method is called with one argument \(pos\), the following steps are taken:

1. Let \(O\) be \texttt{RequireObjectCoercible(this value)}.
2. Let \(S\) be \texttt{ToString(O)}.
3. Let \(position\) be \texttt{ToIntegerOrInfinity(pos)}.
4. Let \(size\) be the length of \(S\).
5. If \(position < 0\) or \(position \geq size\), return \texttt{undefined}.
6. Let \(cp\) be \texttt{! CodePointAt(S, position)}.
7. Return \(cp.[[CodePoint]]\).

NOTE 2

The \texttt{codePointAt} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.4 \texttt{String.prototype.concat ( ...args )}

NOTE 1

When the \texttt{concat} method is called it returns the String value consisting of the code units of the \texttt{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 \texttt{concat} method is called with zero or more arguments, the following steps are taken:

1. Let \(O\) be \texttt{RequireObjectCoercible(this value)}.
2. Let \(S\) be \texttt{ToString(O)}.
3. Let \(R\) be \(S\).
4. For each element \(next\) of \(args\), do
   a. Let \(nextString\) be \texttt{ToString(next)}.
   b. Set \(R\) to the string-concatenation of \(R\) and \(nextString\).
5. Return \(R\).

The "length" property of the \texttt{concat} method is \(1\).

NOTE 2

The \texttt{concat} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.5 \texttt{String.prototype.constructor}

The initial value of \texttt{String.prototype.constructor} is \texttt{%String%}.

22.1.3.6 \texttt{String.prototype.endsWith ( searchString [, endPosition ] )}
The following steps are taken:

1. Let $O$ be \texttt{RequireObjectCoercible(this value)}.
2. Let $S$ be \texttt{ToString($O$)}.
3. Let $isRegExp$ be \texttt{IsRegExp(searchString)}.
4. If $isRegExp$ is \texttt{true}, throw a \texttt{TypeError} exception.
5. Let $searchStr$ be \texttt{ToString(searchString)}.
6. Let $len$ be the length of $S$.
7. If $endPosition$ is \texttt{undefined}, let $pos$ be $len$; else let $pos$ be \texttt{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 \texttt{true}.
11. Let $start$ be $end - searchLength$.
12. If $start < 0$, return \texttt{false}.
13. Let $substring$ be the substring of $S$ from $start$ to $end$.
14. Return \texttt{SameValueNonNumeric(substring, searchStr)}.

\textbf{NOTE 1} Returns \texttt{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 \texttt{false}.

\textbf{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.

\textbf{NOTE 3} The \texttt{endsWith} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

\subsection*{22.1.3.7 String.prototype.includes (searchString [ , position ])}

The \texttt{includes} method takes two arguments, $searchString$ and $position$, and performs the following steps:

1. Let $O$ be \texttt{RequireObjectCoercible(this value)}.
2. Let $S$ be \texttt{ToString($O$)}.
3. Let $isRegExp$ be \texttt{IsRegExp(searchString)}.
4. If $isRegExp$ is \texttt{true}, throw a \texttt{TypeError} exception.
5. Let $searchStr$ be \texttt{ToString(searchString)}.
6. Let $pos$ be \texttt{ToIntegerOrInfinity(position)}.
7. Assert: If $position$ is \texttt{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 \texttt{StringIndexOf($S$, searchStr, start)}.
11. If $index$ is not -1, return \texttt{true}.
12. Return \texttt{false}.
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 $\text{StringIndexOf}(S, \text{searchStr}, \text{start})$.

The `indexOf` method 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.

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 $\operatorname{ToIntegerOrInfinity}(numPos)$.
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 $k + searchLen \leq len$, and for all non-negative integers $j$ such that $j < searchLen$, 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 $\operatorname{RequireObjectCoercible}(this$ value).
2. Let $S$ be $\operatorname{ToString}(O)$.
3. Let $That$ be $\operatorname{ToString}(that)$.

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
   b. If isRegExp is true, then
      i. Let flags be ? Get(regexp, "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 ℝ(? ToLength(maxLength)).
4. Let stringLength be the length of S.
5. If intMaxLength ≠ stringLength, return S.
6. If fillString is undefined, let filler be the String value consisting solely of the code unit 0x0020 (SPACE).
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 +∞, 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
   b. If replacer is not undefined, then
      i. Return ? Call(replacer, searchValue, « O, replaceValue »).
3. Let \textit{string} be ? \texttt{ToString}(O).
4. Let \textit{searchString} be ? \texttt{ToString}(searchValue).
5. Let \textit{functionalReplace} be \texttt{IsCallable}(replaceValue).
6. If \textit{functionalReplace} is \texttt{false}, then
   a. Set \textit{replaceValue} to ? \texttt{ToString}(replaceValue).
7. Let \textit{searchLength} be the length of \textit{searchString}.
8. Let \textit{position} be ! \texttt{StringIndexOf}(\textit{string}, \textit{searchString}, 0).
9. If \textit{position} is -1, return \textit{string}.
10. Let \textit{preserved} be the substring of \textit{string} from 0 to \textit{position}.
11. If \textit{functionalReplace} is \texttt{true}, then
   a. Let \textit{replacement} be ? \texttt{ToString}(! \texttt{Call}(replaceValue, \texttt{undefined}, « searchString, F(position), string »)).
   b. Assert: Type(\textit{replaceValue}) is String.
   c. Let \textit{captures} be a new empty List.
12. Else,
   a. Assert: Type(\textit{replaceValue}) is String.
   b. Let \textit{captures} be a new empty List.
13. Return the string-concatenation of \textit{preserved}, \textit{replacement}, and the substring of \textit{string} from \textit{position} + \textit{searchLength}.

\textbf{NOTE} The \texttt{replace} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

22.1.3.17.1 \texttt{GetSubstitution ( matched, str, position, captures, namedCaptures, replacement )}

The abstract operation \texttt{GetSubstitution} takes arguments \textit{matched}, \textit{str}, \textit{position} (a non-negative integer), \textit{captures}, \textit{namedCaptures}, and \textit{replacement}. It performs the following steps when called:

1. Assert: Type(\textit{matched}) is String.
2. Let \textit{matchLength} be the number of code units in \textit{matched}.
3. Assert: Type(\textit{str}) is String.
4. Let \textit{stringLength} be the number of code units in \textit{str}.
5. Assert: \textit{position} \textless= \textit{stringLength}.
6. Assert: \textit{captures} is a possibly empty List of Strings.
7. Assert: Type(\textit{replacement}) is String.
8. Let \textit{tailPos} be \textit{position} + \textit{matchLength}.
9. Let \textit{m} be the number of elements in \textit{captures}.
10. Let \textit{result} be the String value derived from \textit{replacement} by copying code unit elements from \textit{replacement} to \textit{result}
    while performing replacements as specified in Table 54. These \textdollar{} 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 \textit{result}.
<table>
<thead>
<tr>
<th>Code units</th>
<th>Unicode Characters</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0024, 0x0024</td>
<td>$$ $$</td>
<td>$</td>
</tr>
<tr>
<td>0x0024, 0x0026</td>
<td>$&amp;</td>
<td>matched</td>
</tr>
<tr>
<td>0x0024, 0x0060</td>
<td>$`</td>
<td>The replacement is the substring of <code>str</code> from 0 to <code>position</code>.</td>
</tr>
<tr>
<td>0x0024, 0x0027</td>
<td>$'</td>
<td>If <code>tailPos</code> ≥ <code>stringLength</code>, the replacement is the empty String. Otherwise the replacement is the substring of <code>str</code> from <code>tailPos</code>.</td>
</tr>
<tr>
<td>0x0024, N, N</td>
<td>$n where n is one of 1 2 3 4 5 6 7 8 9 and $n is not followed by a decimal digit</td>
<td>The $n^{th}$ element of <code>captures</code>, where $n$ is a single digit in the range 1 to 9. If $n ≤ m$ and the $n^{th}$ element of <code>captures</code> is <code>undefined</code>, use the empty String instead. If $n &gt; m$, no replacement is done.</td>
</tr>
<tr>
<td>0x0030 ≤ N ≤ 0x0039</td>
<td>$nn where n is one of 0 1 2 3 4 5 6 7 8 9</td>
<td>The $nn^{th}$ element of <code>captures</code>, where $nn$ is a two-digit decimal number in the range 01 to 99. If $nn ≤ m$ and the $nn^{th}$ element of <code>captures</code> is <code>undefined</code>, use the empty String instead. If $nn$ is 00 or $nn &gt; m$, no replacement is done.</td>
</tr>
<tr>
<td>0x0024, 0x003C</td>
<td>$&lt;</td>
<td>1. If <code>namedCaptures</code> is <code>undefined</code>, the replacement text is the String &quot;$&lt;&quot;. 2. Else,  a. Assert: Type(<code>namedCaptures</code>) is Object.  b. Scan until the next &gt; U+003E (GREATER-THAN SIGN).  c. If none is found, the replacement text is the String &quot;$&lt;&quot;.  d. Else,  i. Let <code>groupName</code> be the enclosed substring.  ii. Let <code>capture</code> be ? Get(<code>namedCaptures</code>, <code>groupName</code>).  iii. If <code>capture</code> is <code>undefined</code>, replace the text through &gt; with the empty String.  iv. Otherwise, replace the text through &gt; with ? ToString(<code>capture</code>).</td>
</tr>
<tr>
<td>0x0024</td>
<td>$ in any context that does not match any of the above.</td>
<td>$</td>
</tr>
</tbody>
</table>
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
   b. If `isRegExp` is `true`, then
      i. Let `flags` be ? Get(`searchValue`, "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`).
5. Let `functionalReplace` be IsCallable(`replaceValue`).
6. If `functionalReplace` is `false`, then
   a. Set `replaceValue` to ? ToString(?
   b. Set `functionalReplace` to is `true`, then
      a. Let `replacement` be ? ToString(? Call(`replaceValue`, undefined, « `searchString`, `flags` », `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`.
7. If `endOfLastMatch` < the length of `string`, then
   a. Set `result` to the `string-concatenation` of `result` and the `substring` of `string` from `endOfLastMatch`.
8. 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
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 `-∞`, 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 `-∞`, 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 \( \text{Call(separator, « } \text{O, limit } \text{»).} \)

3. Let \( S \) be \( \text{ToString(}O) \).
4. Let \( A \) be \( \text{ArrayCreate}(0) \).
5. Let \( \text{lengthA} \) be \( 0 \).
6. If \( \text{limit} \) is \text{undefined}, let \( \text{lim} \) be \( 2^{32} - 1 \); else let \( \text{lim} \) be \( \text{R(? ToUint32(} \text{limit} \text{))} \).
7. Let \( R \) be \( \text{ToString(separator)} \).
8. If \( \text{lim} = 0 \), return \( A \).
9. If \( \text{separator} \) is \text{undefined}, then
   a. Perform \( \text{CreateDataPropertyOrThrow}(A, \"0\", S) \).
   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 \( \text{CreateDataPropertyOrThrow}(A, \"0\", S) \).
      b. Return \( A \).
12. Let \( p \) be \( 0 \).
13. Let \( q \) be \( p \).
14. Repeat, while \( q \neq s \),
    a. Let \( e \) be \( \text{SplitMatch}(S, q, R) \).
    b. If \( e \) is not-matched, set \( q \) to \( q + 1 \).
    c. Else,
       i. Assert: \( e \) is a non-negative integer \( \leq s \).
       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 \( \text{CreateDataPropertyOrThrow}(A, ! \text{ToString(F} \text{lengthA}), T) \).
          3. Set \( \text{lengthA} \) to \( \text{lengthA} + 1 \).
          4. If \( \text{lengthA} = \text{lim} \), 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 \( \text{CreateDataPropertyOrThrow}(A, ! \text{ToString(F} \text{lengthA}), T) \).
17. Return \( A \).

\textbf{NOTE 1} \hspace{1cm} \text{The value of} separator \text{ may be an empty String. In this case,} separator \text{ 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 \text{ 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.}

\text{If the} this \text{ value is (or converts to) the empty String, the result depends on whether} separator \text{ 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.}

\text{If} separator \text{ is undefined, then the result array contains just one String, which is the} this \text{ value (converted to a String). If} limit \text{ is not} undefined, \text{then the output array is truncated so that it contains no more than} limit \text{ 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)`.
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)

569
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.toUpperCasetoUpperCase()`

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()`

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 \( ? \text{TrimString}(S, \text{end}) \).

**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 \( ? \text{TrimString}(S, \text{start}) \).

**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 \( ? \text{thisStringValue}(\text{this value}) \).

### 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 \( ? \text{RequireObjectCoercible}(\text{this value}) \).
2. Let \( s \) be \( ? \text{ToString}(O) \).
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 ? \text{Yield}(resultString).
   d. Return undefined.
4. Return \( ! \text{CreateIteratorFromClosure}(\text{closure}, "%\text{StringIteratorPrototype}\%", %\text{StringIteratorPrototype}\%) \).

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

\[ Pattern_{[U, N]} :: \]
\[ \text{Disjunction}_{[?U, ?N]} \]

\[ \text{Disjunction}_{[U, N]} :: \]
\[ \text{Alternative}_{[?U, ?N]} \]
\[ \text{Alternative}_{[?U, ?N]} | \text{Disjunction}_{[?U, ?N]} \]

\[ \text{Alternative}_{[U, N]} :: \]
\[ \{ \text{empty} \} \]
\[ \text{Alternative}_{[?U, ?N]} \text{ Term}_{[?U, ?N]} \]

\[ \text{Term}_{[U, N]} :: \]
\[ \text{Assertion}_{[?U, ?N]} \]
\[ \text{Atom}_{[?U, ?N]} \]
\[ \text{Atom}_{[?U, ?N]} \text{ Quantifier} \]

\[ \text{Assertion}_{[U, N]} :: \]
\[ ^ \]
\[ \$ \]
\[ \backslash b \]
\[ \backslash B \]
\[ ( ? = \text{Disjunction}_{[?U, ?N]} ) \]
\[ ( ? ! \text{Disjunction}_{[?U, ?N]} ) \]
\[ ( ? <= \text{Disjunction}_{[?U, ?N]} ) \]
\[ ( ? <! \text{Disjunction}_{[?U, ?N]} ) \]

\[ \text{Quantifier} :: \]
\[ \text{QuantifierPrefix} \]
\[ \text{QuantifierPrefix} ? \]

\[ \text{QuantifierPrefix} :: \]
\[ * \]
\[ + \]
\[ ? \]
\[ \{ \text{DecimalDigits}_{[-\text{Sep}]} \} \]
\[ \{ \text{DecimalDigits}_{[-\text{Sep}]} , \} \]
\[ \{ \text{DecimalDigits}_{[-\text{Sep}]} , \text{DecimalDigits}_{[-\text{Sep}]} \} \]

\[ \text{Atom}_{[U, N]} :: \]
\[ \text{PatternCharacter} \]
\[ . \]
\ AtomEscape \[7u, 7n\]
CharacterClass \[7u\]
( GroupSpecifier \[7u\] Disjunction \[7u, 7n\] )
( ? : Disjunction \[7u, 7n\] )

SyntaxCharacter :: one of
  ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
  SourceCharacter but not SyntaxCharacter

AtomEscape \[u, n\] ::
  DecimalEscape
  CharacterClassEscape \[7u\]
  CharacterEscape \[7u\]
  \[^-N] k GroupName \[7u\]

CharacterEscape \[u\] ::
  ControlEscape
c ControlLetter
  \[0 \text{ lookahead } \notin \text{ DecimalDigit}\]
  HexEscapeSequence
  RegExpUnicodeEscapeSequence \[7u\]
  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 \[7u\]

GroupName \[u\] ::
  < RegExpIdentifierName \[7u\] >

RegExpIdentifierName \[u\] ::
  RegExpIdentifierStart \[7u\]
  RegExpIdentifierName \[7u\] RegExpIdentifierPart \[7u\]

RegExpIdentifierStart \[u\] ::
  UnicodeIDStart
  $
RegExpUnicodeEscapeSequence

-
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] \( p\{ \text{UnicodePropertyValueExpression} \} \)
[+U] \( p\{ \text{UnicodePropertyValueExpression} \} \)

\text{UnicodePropertyValueExpression} ::
\text{UnicodePropertyName} = \text{UnicodePropertyValue}
\text{LoneUnicodePropertyNameOrValue}

\text{UnicodePropertyName} ::
\text{UnicodePropertyNameCharacters}

\text{UnicodePropertyNameCharacters} ::
\text{UnicodePropertyNameCharacter} \text{UnicodePropertyNameCharacters}_{\text{opt}}

\text{UnicodePropertyValue} ::
\text{UnicodePropertyValueCharacters}

\text{LoneUnicodePropertyNameOrValue} ::
\text{UnicodePropertyValueCharacters}

\text{UnicodePropertyValueCharacters} ::
\text{UnicodePropertyValueCharacter} \text{UnicodePropertyValueCharacters}_{\text{opt}}

\text{UnicodePropertyValueCharacter} ::
\text{UnicodePropertyNameCharacter}
\text{DecimalDigit}

\text{UnicodePropertyNameCharacter} ::
\text{ControlLetter}
-

\text{CharacterClass}_{[\text{U}]} ::
\{ \text{lookahead } \neq \} \text{ClassRanges}_{[\text{?U}]} \}
\{ \^ \text{ClassRanges}_{[\text{?U}]} \}

\text{ClassRanges}_{[\text{U}]} ::
[\text{empty}]
\text{NonemptyClassRanges}_{[\text{?U}]}$

\text{NonemptyClassRanges}_{[\text{U}]} ::
\text{ClassAtom}_{[\text{?U}]}
\text{ClassAtom}_{[\text{?U}]} \text{NonemptyClassRangesNoDash}_{[\text{?U}]}
\text{ClassAtom}_{[\text{?U}]} - \text{ClassAtom}_{[\text{?U}]} \text{ClassRanges}_{[\text{?U}]}
NonemptyClassRangesNoDash[u]  ::
  ClassAtom[7u]
  ClassAtomNoDash[7u]  NonemptyClassRangesNoDash[7u]
  ClassAtomNoDash[7u]  - ClassAtom[7u]  ClassRanges[7u]

ClassAtom[7u]  ::
  -
  ClassAtomNoDash[7u]

ClassAtomNoDash[7u]  ::
  SourceCharacter  but not one of \ or ] or -
  \ ClassEscape[7u]

ClassEscape[7u]  ::
  b
  [+U] -
  CharacterClassEscape[7u]
  CharacterEscape[7u]

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 ≥ 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 CapturingGroupName 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 \texttt{IsCharacterClass} of the first \texttt{ClassAtom} is \texttt{false} and \texttt{IsCharacterClass} of the second \texttt{ClassAtom} is \texttt{false} and the CharacterValue of the first \texttt{ClassAtom} is larger than the CharacterValue of the second \texttt{ClassAtom}.

\texttt{NonemptyClassRangesNoDash} :: \texttt{ClassAtomNoDash} - \texttt{ClassAtom} \texttt{ClassRanges}

- It is a Syntax Error if \texttt{IsCharacterClass} of \texttt{ClassAtomNoDash} is \texttt{true} or \texttt{IsCharacterClass} of \texttt{ClassAtom} is \texttt{true}.
- It is a Syntax Error if \texttt{IsCharacterClass} of \texttt{ClassAtomNoDash} is \texttt{false} and \texttt{IsCharacterClass} of \texttt{ClassAtom} is \texttt{false} and the CharacterValue of \texttt{ClassAtomNoDash} is larger than the CharacterValue of \texttt{ClassAtom}.

\texttt{RegExpIdentifierStart\_{U}} :: \ RegExpUnicodeEscapeSequence\_{[+U]}

- It is a Syntax Error if the CharacterValue of \texttt{RegExpUnicodeEscapeSequence} is not the code point value of "$", ",", or some code point matched by the \texttt{UnicodeIDStart} lexical grammar production.

\texttt{RegExpIdentifierStart\_{U}} :: \ UnicodeLeadSurrogate \ UnicodeTrailSurrogate

- It is a Syntax Error if the result of performing \texttt{UTF16SurrogatePairToCodePoint} on the two code points matched by \texttt{UnicodeLeadSurrogate} and \texttt{UnicodeTrailSurrogate} respectively is not matched by the \texttt{UnicodeIDStart} lexical grammar production.

\texttt{RegExpIdentifierPart\_{U}} :: \ RegExpUnicodeEscapeSequence\_{[+U]}

- It is a Syntax Error if the CharacterValue of \texttt{RegExpUnicodeEscapeSequence} is not the code point value of "$", ",", <\texttt{ZWNJ}>, <\texttt{ZWJ}>, or some code point matched by the \texttt{UnicodeIDContinue} lexical grammar production.

\texttt{RegExpIdentifierPart\_{U}} :: \ UnicodeLeadSurrogate \ UnicodeTrailSurrogate

- It is a Syntax Error if the result of performing \texttt{UTF16SurrogatePairToCodePoint} on the two code points matched by \texttt{UnicodeLeadSurrogate} and \texttt{UnicodeTrailSurrogate} respectively is not matched by the \texttt{UnicodeIDContinue} lexical grammar production.

\texttt{UnicodePropertyValueExpression} :: \ UnicodePropertyName = \ UnicodePropertyValue

- It is a Syntax Error if the List of Unicode code points that is SourceText of \texttt{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 \texttt{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 \texttt{UnicodePropertyName} listed in the “Property value and aliases” column of the corresponding tables Table 58 or Table 59.

\texttt{UnicodePropertyValueExpression} :: \ LoneUnicodePropertyNameOrValue

- It is a Syntax Error if the List of Unicode code points that is SourceText of \texttt{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
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 1 This section is amended in B.1.4.3.

ClassAtom :: -
ClassAtomNoDash :: SourceCharacter but not one of \( \backslash \) 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 \( \backslash \) 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>
<thead>
<tr>
<th>ControlEscape</th>
<th>Code Point Value</th>
<th>Code Point</th>
<th>Unicode Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>9</td>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>U+000A</td>
<td>LINE FEED (LF)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>v</td>
<td>11</td>
<td>U+000B</td>
<td>LINE TABULATION</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>f</td>
<td>12</td>
<td>U+000C</td>
<td>FORM FEED (FF)</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>r</td>
<td>13</td>
<td>U+000D</td>
<td>CARRIAGE RETURN (CR)</td>
<td>&lt;CR&gt;</td>
</tr>
</tbody>
</table>

**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 ∉ 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.
1. Let $ch$ be the code point matched by $\text{IdentityEscape}$.
2. Return the code point value of $ch$.

### 22.2.1.5 Static Semantics: SourceText

$\text{UnicodePropertyNameCharacters} :: \text{UnicodePropertyNameCharacter} \; \text{UnicodePropertyNameCharacters}_{\text{opt}}$

$\text{UnicodePropertyValueCharacters} :: \text{UnicodePropertyValueCharacter} \; \text{UnicodePropertyValueCharacters}_{\text{opt}}$

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

$\text{RegExpIdentifierName}_{[u]} ::$

$\text{RegExpIdentifierStart} \; \text{RegExpIdentifierName}_{[u]} \; \text{RegExpIdentifierPart}_{[u]}$

1. Let $idText$ be the source text matched by $\text{RegExpIdentifierName}$.
2. Let $idTextUnescaped$ be the result of replacing any occurrences of $\text{RegExpUnicodeEscapeSequence}$ in $idText$ with the code point represented by the $\text{RegExpUnicodeEscapeSequence}$.
3. Return $\text{CodePointsToString}(idTextUnescaped)$.

### 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.
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 \( \text{Input}[n] \) means the \( n \)th character of \( \text{Input} \), where \( n \) can range between 0 (inclusive) and \( \text{InputLength} \) (exclusive).
- **InputLength** is the number of characters in \( \text{Input} \).
- **NcapturingParens** is the total number of left-capturing parentheses (i.e. the total number of \( \text{Atom} :: ( \text{GroupSpecifier} \mid \text{Disjunction} ) \) Parse Nodes) in the pattern. A left-capturing parenthesis is any \( ( \) pattern character that is matched by the \( \text{Terminal} \) of the \( \text{Atom} :: ( \text{GroupSpecifier} \mid \text{Disjunction} ) \) production.
- **DotAll** is \textbf{true} if the RegExp object's [[OriginalFlags]] internal slot contains "s" and otherwise is \textbf{false}.
- **IgnoreCase** is \textbf{true} if the RegExp object's [[OriginalFlags]] internal slot contains "i" and otherwise is \textbf{false}.
- **Multiline** is \textbf{true} if the RegExp object's [[OriginalFlags]] internal slot contains "m" and otherwise is \textbf{false}.
- **Unicode** is \textbf{true} if the RegExp object's [[OriginalFlags]] internal slot contains "u" and otherwise is \textbf{false}.
- **WordCharacters** is the mathematical set that is the union of all sixty-three characters in "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_" (letters, numbers, and \( \text{U+005F} \) (LOW LINE) in the Unicode Basic Latin block) and all characters \( c \) for which \( \text{Canonicalize}(c) \) is. **WordCharacters** cannot contain more than sixty-three characters unless **Unicode** and **IgnoreCase** are both \textbf{true}.

Furthermore, the descriptions below use the following internal data structures:

- A CharSet is a mathematical set of characters. When the **Unicode** flag is \textbf{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 \( (\text{endIndex}, \text{captures}) \) where \( \text{endIndex} \) is an integer and \( \text{captures} \) is a List of **NcapturingParens** values. States are used to represent partial match states in the regular expression matching algorithms. The \( \text{endIndex} \) is one plus the index of the last input character matched so far by the pattern, while \( \text{captures} \) holds the results of capturing parentheses. The \( n \)th element of \( \text{captures} \) is either a List of characters that represents the value obtained by the \( n \)th set of capturing parentheses or \textbf{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 \textit{direction}.

The production \texttt{Disjunction :: Alternative} evaluates as follows:

1. Evaluate \texttt{Alternative} with argument \texttt{direction} to obtain a Matcher \texttt{m}.
2. Return \texttt{m}.

The production \texttt{Disjunction :: Alternative | Disjunction} evaluates as follows:

1. Evaluate \texttt{Alternative} with argument \texttt{direction} to obtain a Matcher \texttt{m1}.
2. Evaluate \texttt{Disjunction} with argument \texttt{direction} to obtain a Matcher \texttt{m2}.
3. Return a new Matcher with parameters \((x, c)\) that captures \texttt{m1} and \texttt{m2} and performs the following steps when called:
   a. \textbf{Assert}: \(x\) is a State.
   b. \textbf{Assert}: \(c\) is a Continuation.
   c. Let \(r\) be \texttt{m1}(\(x, c\)).
   d. If \(r\) is not failure, return \(r\).
   e. Return \texttt{m2}(\(x, c\)).

\textbf{NOTE} The \texttt{\mid} regular expression operator separates two alternatives. The pattern first tries to match the left \texttt{Alternative} (followed by the sequel of the regular expression); if it fails, it tries to match the right \texttt{Disjunction} (followed by the sequel of the regular expression). If the left \texttt{Alternative}, the right \texttt{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 \texttt{Alternative}. If choices in the left \texttt{Alternative} are exhausted, the right \texttt{Disjunction} is tried instead of the left \texttt{Alternative}. Any capturing parentheses inside a portion of the pattern skipped by \texttt{\mid} produce \texttt{undefined} values instead of Strings. Thus, for example,

\begin{verbatim}
/alab/.exec("abc")
\end{verbatim}

returns the result "a" and not "ab". Moreover,

\begin{verbatim}
/((a)|(ab))((c)|(bc))/.exec("abc")
\end{verbatim}

returns the array

\begin{verbatim}
["abc", "a", "a", undefined, "bc", undefined, "bc"]
\end{verbatim}

and not

\begin{verbatim}
["abc", "ab", undefined, "ab", "c", "c", undefined]
\end{verbatim}

The order in which the two alternatives are tried is independent of the value of \textit{direction}.

22.2.2.4 Alternative

With parameter \textit{direction}.

The production \texttt{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 $\text{Alternative} :: \text{Alternative Term}$ evaluates as follows:

1. Evaluate $\text{Alternative}$ with argument $\text{direction}$ to obtain a Matcher $m1$.
2. Evaluate $\text{Term}$ with argument $\text{direction}$ to obtain a Matcher $m2$.
3. If $\text{direction} = 1$, 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: $\text{direction}$ 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 $\text{Terms}$ try to simultaneously match consecutive portions of $\text{Input}$. When $\text{direction} = 1$, if the left $\text{Alternative}$, the right $\text{Term}$, 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 $\text{Term}$, and all choices in the right $\text{Term}$ are tried before moving on to the next choice in the left $\text{Alternative}$. When $\text{direction} = -1$, the evaluation order of $\text{Alternative}$ and $\text{Term}$ are reversed.

22.2.2.5 $\text{Term}$

With parameter $\text{direction}$.

The production $\text{Term} :: \text{Assertion}$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $\text{Assertion}$.

NOTE
The resulting Matcher is independent of $\text{direction}$.

The production $\text{Term} :: \text{Atom}$ 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 $min = 0$, 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 $c(x)$. 

588
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.

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\text{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.

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:

```
"aaaaaaaaaaaa,aaaaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/, "$1")
```

which returns the gcd in unary notation "aaaaa".
22.2.6 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.
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 \(e = InputLength\), or if \(.Multiline\) is \(true\) and the character \(Input[e]\) is one of \(LineTerminator\), then
      
      i. Return \(c(x)\).
   
   e. Return failure.

The production \(Assertion ::= \backslash b\) 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 \(Assertion ::= \backslash b\) 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 \( \text{InputLength} \), return \text{false}.
2. Let \( c \) be the character \( \text{Input}[e] \).
3. If \( c \) is in \( \text{WordCharacters} \), return \text{true}.
4. Return \text{false}.

22.2.2.7 Quantifier

The production \( \text{Quantifier :: QuantifierPrefix} \) evaluates as follows:

1. Evaluate \( \text{QuantifierPrefix} \) to obtain the two results: an \text{integer \( min \)} and an \text{integer \( (or \ +\infty) \ max \)}.
2. Return the three results \( min, max, \) and \text{true}.

The production \( \text{Quantifier :: QuantifierPrefix} \ ? \) evaluates as follows:

1. Evaluate \( \text{QuantifierPrefix} \) to obtain the two results: an \text{integer \( min \)} and an \text{integer \( (or \ +\infty) \ max \)}.
2. Return the three results \( min, max, \) and \text{false}.

The production \( \text{QuantifierPrefix :: \*} \) evaluates as follows:

1. Return the two results 0 and \( +\infty \).

The production \( \text{QuantifierPrefix :: \+} \) evaluates as follows:

1. Return the two results 1 and \( +\infty \).

The production \( \text{QuantifierPrefix :: ?} \) evaluates as follows:

1. Return the two results 0 and 1.

The production \( \text{QuantifierPrefix :: \{ DecimalDigits \}} \) evaluates as follows:

1. Let \( i \) be the MV of \( \text{DecimalDigits} \) (see 12.8.3).
2. Return the two results \( i \) and \( i \).

The production \( \text{QuantifierPrefix :: \{ DecimalDigits , \}} \) evaluates as follows:

1. Let \( i \) be the MV of \( \text{DecimalDigits} \).
2. Return the two results \( i \) and \( +\infty \).

The production \( \text{QuantifierPrefix :: \{ DecimalDigits , DecimalDigits \}} \) evaluates as follows:

1. Let \( i \) be the MV of the first \( \text{DecimalDigits} \).
2. Let \( j \) be the MV of the second \( \text{DecimalDigits} \).
3. Return the two results \( i \) and \( j \).

22.2.2.8 Atom

With parameter \text{direction}.

The production \( \text{Atom :: PatternCharacter} \) evaluates as follows:

1. Let \( ch \) be the character matched by \( \text{PatternCharacter} \).
2. Let \( A \) be a one-element CharSet containing the character \( ch \).
3. Return ! \text{CharacterSetMatcher}(A, \text{false}, \text{direction}).
The production \textit{Atom} :: \textit{.} evaluates as follows:

1. Let \( A \) be the CharSet of all characters.
2. If \textit{DotAll} is not true, then
   a. Remove from \( A \) all characters corresponding to a code point on the right-hand side of the \textit{LineTerminator} production.
3. Return \( ! \) \text{CharacterSetMatcher}(A, \text{false}, \text{direction}).

The production \textit{Atom} :: \( \backslash \text{AtomEscape} \) evaluates as follows:

1. Return the Matcher that is the result of evaluating \textit{AtomEscape} with argument \textit{direction}.

The production \textit{Atom} :: \textit{CharacterClass} evaluates as follows:

1. Evaluate \textit{CharacterClass} to obtain a CharSet \( A \) and a Boolean \textit{invert}.
2. Return \( ! \) \text{CharacterSetMatcher}(A, \text{invert}, \text{direction}).

The production \textit{Atom} :: ( \textit{GroupSpecifier} \textit{Disjunction} ) evaluates as follows:

1. Evaluate \textit{Disjunction} with argument \textit{direction} to obtain a Matcher \( m \).
2. Let \textit{parenIndex} be the number of left-capturing parentheses in the entire regular expression that occur to the left of this \textit{Atom}. This is the total number of \textit{Atom} :: ( \textit{GroupSpecifier} \textit{Disjunction} ) Parse Nodes prior to or enclosing this \textit{Atom}.
3. Return a new Matcher with parameters \((x, c)\) that captures \textit{direction}, \(m\), and \textit{parenIndex} and performs the following steps when called:
   a. \text{Assert}: \( x \) is a State.
   b. \text{Assert}: \( c \) is a Continuation.
   c. Let \( d \) be a new Continuation with parameters \((y)\) that captures \(x, c, \text{direction}, \) and \textit{parenIndex} and performs the following steps when called:
      i. \text{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 \textit{direction} = 1, then
         1. \text{Assert}: \( xe \leq ye \).
         2. Let \( s \) be a List whose elements are the characters of \textit{Input} at indices \( xe \) (inclusive) through \( ye \) (exclusive).
      vi. Else,
         1. \text{Assert}: \textit{direction} is -1.
         2. \text{Assert}: \( ye \leq xe \).
         3. Let \( s \) be a List whose elements are the characters of \textit{Input} at indices \( ye \) (inclusive) through \( xe \) (exclusive).
      vii. Set \text{cap}[\text{parenIndex} + 1] to \textit{s}.
      viii. Let \( z \) be the State \((ye, cap)\).
      ix. Return \text{c}(z).
   d. Return \( m(x, d) \).

The production \textit{Atom} :: ( \textit{?} : \textit{Disjunction} ) evaluates as follows:

1. Return the Matcher that is the result of evaluating \textit{Disjunction} with argument \textit{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\(\langle cp \rangle\), according to the Unicode Default Case Conversion algorithm.
6. Let \(uStr\) be \(\text{! CodePointsToString}(u)\).
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 \(ch \geq 128\) and the numeric value of \(cu < 128\), return \(ch\).
10. Return \(cu\).

**NOTE 1** Parentheses of the form \((\text{Disjunction})\) serve both to group the components of the \text{Disjunction} pattern together and to save the result of the match. The result can be used either in a backreference (\(\backslash\) 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 \((? : \text{Disjunction})\) instead.
The form $(?= \text{Disjunction})$ 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 $(?= \text{form})$ (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("baabac")
```

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("baabac")
```

This expression returns

```
["aba", "a"]
```

and not:

```
["aaaba", "a"]
```

The form $(?! \text{Disjunction})$ 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\2)c)\2(.*)/.exec("baabac")
```

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:

```
["baabac", "ba", undefined, "abaac"]
```
NOTE 4 In case-insensitive 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, Ŧ (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.

### 22.2.8.3 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

<table>
<thead>
<tr>
<th>Property name and aliases</th>
<th>Canonical property name</th>
</tr>
</thead>
<tbody>
<tr>
<td>General_Category</td>
<td>General_Category</td>
</tr>
<tr>
<td>gc</td>
<td></td>
</tr>
<tr>
<td>Script</td>
<td>Script</td>
</tr>
<tr>
<td>sc</td>
<td></td>
</tr>
<tr>
<td>Script_Extensions</td>
<td>Script_Extensions</td>
</tr>
<tr>
<td>scx</td>
<td></td>
</tr>
</tbody>
</table>

### Table 57: Binary Unicode property aliases and their canonical property names

<table>
<thead>
<tr>
<th>Property name and aliases</th>
<th>Canonical property name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>ASCII</td>
</tr>
<tr>
<td>ASCII_Hex_Digit</td>
<td>ASCII_Hex_Digit</td>
</tr>
<tr>
<td>AHex</td>
<td></td>
</tr>
<tr>
<td>Alphabetic</td>
<td>Alphabetic</td>
</tr>
<tr>
<td>Character Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Alpha</td>
<td>Any</td>
</tr>
<tr>
<td>Any</td>
<td>Assigned</td>
</tr>
<tr>
<td>Assigned</td>
<td>Bidi_Control</td>
</tr>
<tr>
<td>Bidi_Control</td>
<td>Bidi_Mirrored</td>
</tr>
<tr>
<td>Bidi_Mirrored</td>
<td>Case_Ignorable</td>
</tr>
<tr>
<td>Case_Ignorable</td>
<td>Cased</td>
</tr>
<tr>
<td>Cased</td>
<td>Changes_When_Casefolded</td>
</tr>
<tr>
<td>Changes_When_Casefolded</td>
<td>Changes_When_Casemapped</td>
</tr>
<tr>
<td>Changes_When_Casemapped</td>
<td>Changes_When_Lowercased</td>
</tr>
<tr>
<td>Changes_When_Lowercased</td>
<td>Changes_When_NFKC_Casefolded</td>
</tr>
<tr>
<td>Changes_When_NFKC_Casefolded</td>
<td>Changes_When_Titlecased</td>
</tr>
<tr>
<td>Changes_When_Titlecased</td>
<td>Changes_When_Uppercased</td>
</tr>
<tr>
<td>Changes_When_Uppercased</td>
<td>Dash</td>
</tr>
<tr>
<td>Dash</td>
<td>Default_Ignorable_Code_Point</td>
</tr>
<tr>
<td>Default_Ignorable_Code_Point</td>
<td>Deprecated</td>
</tr>
<tr>
<td>Deprecated</td>
<td>Dep</td>
</tr>
<tr>
<td>Dep</td>
<td>Diacritic</td>
</tr>
<tr>
<td>Diacritic</td>
<td>Dia</td>
</tr>
<tr>
<td>Dia</td>
<td>Emoji</td>
</tr>
<tr>
<td>Emoji</td>
<td>Emoji_Component</td>
</tr>
<tr>
<td>Emoji_Component</td>
<td>EComp</td>
</tr>
<tr>
<td>EComp</td>
<td>Emoji_Marker</td>
</tr>
<tr>
<td>Emoji_Marker</td>
<td>EMod</td>
</tr>
<tr>
<td>EMod</td>
<td>Emoji_Marker_Base</td>
</tr>
<tr>
<td>Emoji_Marker_Base</td>
<td>EBase</td>
</tr>
<tr>
<td>EBase</td>
<td></td>
</tr>
<tr>
<td>Emoji_Presentation</td>
<td>Emoji_Presentation</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>EPres</td>
<td>EPres</td>
</tr>
<tr>
<td>Extended_Pictographic</td>
<td>Extended_Pictographic</td>
</tr>
<tr>
<td>ExtPict</td>
<td>ExtPict</td>
</tr>
<tr>
<td>Extender</td>
<td>Extender</td>
</tr>
<tr>
<td>Ext</td>
<td>Ext</td>
</tr>
<tr>
<td>Grapheme_Base</td>
<td>Grapheme_Base</td>
</tr>
<tr>
<td>Gr_Base</td>
<td>Gr_Base</td>
</tr>
<tr>
<td>Grapheme_Extend</td>
<td>Grapheme_Extend</td>
</tr>
<tr>
<td>Gr_Ext</td>
<td>Gr_Ext</td>
</tr>
<tr>
<td>Hex_Digit</td>
<td>Hex_Digit</td>
</tr>
<tr>
<td>Hex</td>
<td>Hex</td>
</tr>
<tr>
<td>IDS_Binary_Operator</td>
<td>IDS_Binary_Operator</td>
</tr>
<tr>
<td>IDSB</td>
<td>IDSB</td>
</tr>
<tr>
<td>IDS_Trinary_Operator</td>
<td>IDS_Trinary_Operator</td>
</tr>
<tr>
<td>IDST</td>
<td>IDST</td>
</tr>
<tr>
<td>ID_Continue</td>
<td>ID_Continue</td>
</tr>
<tr>
<td>IDC</td>
<td>IDC</td>
</tr>
<tr>
<td>ID_Start</td>
<td>ID_Start</td>
</tr>
<tr>
<td>IDS</td>
<td>IDS</td>
</tr>
<tr>
<td>Ideographic</td>
<td>Ideographic</td>
</tr>
<tr>
<td>Ideo</td>
<td>Ideo</td>
</tr>
<tr>
<td>Join_Control</td>
<td>Join_Control</td>
</tr>
<tr>
<td>Join_C</td>
<td>Join_C</td>
</tr>
<tr>
<td>Logical_Order_Exception</td>
<td>Logical_Order_Exception</td>
</tr>
<tr>
<td>LOE</td>
<td>LOE</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Lowercase</td>
</tr>
<tr>
<td>Lower</td>
<td>Lower</td>
</tr>
<tr>
<td>Math</td>
<td>Math</td>
</tr>
<tr>
<td>Noncharacter_Code_Point</td>
<td>Noncharacter_Code_Point</td>
</tr>
<tr>
<td>NChar</td>
<td>NChar</td>
</tr>
<tr>
<td>Pattern_Syntax</td>
<td>Pattern_Syntax</td>
</tr>
<tr>
<td>Pat_Syn</td>
<td>Pat_Syn</td>
</tr>
<tr>
<td>Pattern_White_Space</td>
<td>Pattern_White_Space</td>
</tr>
<tr>
<td>Pat_WS</td>
<td>Pat_WS</td>
</tr>
<tr>
<td>Quotation_Mark</td>
<td>Quotation_Mark</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>QMark</th>
<th>Radical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radical_Indicator</td>
<td>Regional_Indicator</td>
</tr>
<tr>
<td>RI</td>
<td></td>
</tr>
<tr>
<td>Sentence_Terminal</td>
<td>Sentence_Terminal</td>
</tr>
<tr>
<td>STerm</td>
<td></td>
</tr>
<tr>
<td>Soft_Dotted</td>
<td>Soft_Dotted</td>
</tr>
<tr>
<td>SD</td>
<td></td>
</tr>
<tr>
<td>Terminal_Punctuation</td>
<td>Terminal_Punctuation</td>
</tr>
<tr>
<td>Term</td>
<td></td>
</tr>
<tr>
<td>Unified_Ideograph</td>
<td>Unified_Ideograph</td>
</tr>
<tr>
<td>UIdeo</td>
<td></td>
</tr>
<tr>
<td>Uppercase</td>
<td>Uppercase</td>
</tr>
<tr>
<td>Upper</td>
<td></td>
</tr>
<tr>
<td>Variation_Selector</td>
<td>Variation_Selector</td>
</tr>
<tr>
<td>VS</td>
<td></td>
</tr>
<tr>
<td>White_Space</td>
<td>White_Space</td>
</tr>
<tr>
<td>space</td>
<td></td>
</tr>
<tr>
<td>XID_Continue</td>
<td>XID_Continue</td>
</tr>
<tr>
<td>XIDC</td>
<td></td>
</tr>
<tr>
<td>XID_Start</td>
<td>XID_Start</td>
</tr>
<tr>
<td>XIDS</td>
<td></td>
</tr>
</tbody>
</table>
For example, `Xpeo` and `Old_Persian` are valid `Script_Extensions` values, but `xpeo` and `Old Persian` aren't.

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`  

<table>
<thead>
<tr>
<th>Property value and aliases</th>
<th>Canonical property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cased_Letter</td>
<td>Cased_Letter</td>
</tr>
<tr>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>Close_Punctuation</td>
<td>Close_Punctuation</td>
</tr>
<tr>
<td>Pe</td>
<td></td>
</tr>
<tr>
<td>Connector_Punctuation</td>
<td>Connector_Punctuation</td>
</tr>
<tr>
<td>Pc</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>Control</td>
</tr>
<tr>
<td>Cc</td>
<td></td>
</tr>
<tr>
<td>cntrl</td>
<td></td>
</tr>
<tr>
<td>Currency_Symbol</td>
<td>Currency_Symbol</td>
</tr>
<tr>
<td>Sc</td>
<td></td>
</tr>
<tr>
<td>Dash_Punctuation</td>
<td>Dash_Punctuation</td>
</tr>
<tr>
<td>Pd</td>
<td></td>
</tr>
<tr>
<td>Decimal_Number</td>
<td>Decimal_Number</td>
</tr>
<tr>
<td>Nd</td>
<td></td>
</tr>
<tr>
<td>digit</td>
<td></td>
</tr>
<tr>
<td>Enclosing_Mark</td>
<td>Enclosing_Mark</td>
</tr>
<tr>
<td>Me</td>
<td></td>
</tr>
<tr>
<td>Final_Punctuation</td>
<td>Final_Punctuation</td>
</tr>
<tr>
<td>Pf</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Format</td>
</tr>
<tr>
<td>Cf</td>
<td></td>
</tr>
<tr>
<td>Initial_Punctuation</td>
<td>Initial_Punctuation</td>
</tr>
<tr>
<td>Pi</td>
<td></td>
</tr>
<tr>
<td>Letter</td>
<td>Letter</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Letter_Number</td>
<td>Letter_Number</td>
</tr>
<tr>
<td>NL</td>
<td></td>
</tr>
<tr>
<td>Line_Separator</td>
<td>Line_Separator</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Zl</td>
<td>Zl</td>
</tr>
<tr>
<td>Lowercase_Letter</td>
<td>Lowercase_Letter</td>
</tr>
<tr>
<td>Ll</td>
<td>Ll</td>
</tr>
<tr>
<td>Mark</td>
<td>Mark</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Combining_Mark</td>
<td>Combining_Mark</td>
</tr>
<tr>
<td>Math_Symbol</td>
<td>Math_Symbol</td>
</tr>
<tr>
<td>Sm</td>
<td>Sm</td>
</tr>
<tr>
<td>Modifier_Letter</td>
<td>Modifier_Letter</td>
</tr>
<tr>
<td>Lm</td>
<td>Lm</td>
</tr>
<tr>
<td>Modifier_Symbol</td>
<td>Modifier_Symbol</td>
</tr>
<tr>
<td>Sk</td>
<td>Sk</td>
</tr>
<tr>
<td>Nonspacing_Mark</td>
<td>Nonspacing_Mark</td>
</tr>
<tr>
<td>Mn</td>
<td>Mn</td>
</tr>
<tr>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Open_Punctuation</td>
<td>Open_Punctuation</td>
</tr>
<tr>
<td>Ps</td>
<td>Ps</td>
</tr>
<tr>
<td>Other</td>
<td>Other</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Other_Letter</td>
<td>Other_Letter</td>
</tr>
<tr>
<td>Lo</td>
<td>Lo</td>
</tr>
<tr>
<td>Other_Number</td>
<td>Other_Number</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Other_Punctuation</td>
<td>Other_Punctuation</td>
</tr>
<tr>
<td>Po</td>
<td>Po</td>
</tr>
<tr>
<td>Other_Symbol</td>
<td>Other_Symbol</td>
</tr>
<tr>
<td>So</td>
<td>So</td>
</tr>
<tr>
<td>Paragraph_Separator</td>
<td>Paragraph_Separator</td>
</tr>
<tr>
<td>Zp</td>
<td>Zp</td>
</tr>
<tr>
<td>Private_Use</td>
<td>Private_Use</td>
</tr>
<tr>
<td>Co</td>
<td>Co</td>
</tr>
<tr>
<td>Punctuation</td>
<td>Punctuation</td>
</tr>
<tr>
<td>punct</td>
<td>punct</td>
</tr>
<tr>
<td>Property value and aliases</td>
<td>Canonical property value</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Adlam</td>
<td>Adlam</td>
</tr>
<tr>
<td>Adlm</td>
<td>Adlm</td>
</tr>
<tr>
<td>Ahom</td>
<td>Ahom</td>
</tr>
<tr>
<td>Anatolian_Hieroglyphs</td>
<td>Anatolian_Hieroglyphs</td>
</tr>
<tr>
<td>Hluw</td>
<td></td>
</tr>
<tr>
<td>Arabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>Arab</td>
<td></td>
</tr>
<tr>
<td>Armenian</td>
<td>Armenian</td>
</tr>
<tr>
<td>Armn</td>
<td></td>
</tr>
<tr>
<td>Avestan</td>
<td>Avestan</td>
</tr>
<tr>
<td>Avst</td>
<td></td>
</tr>
<tr>
<td>Balinese</td>
<td>Balinese</td>
</tr>
<tr>
<td>Bali</td>
<td></td>
</tr>
<tr>
<td>Bamum</td>
<td>Bamum</td>
</tr>
<tr>
<td>Bamu</td>
<td></td>
</tr>
<tr>
<td>Bassa_Vah</td>
<td>Bassa_Vah</td>
</tr>
<tr>
<td>Bass</td>
<td></td>
</tr>
</tbody>
</table>

Table 59: Value aliases and canonical values for the Unicode properties **Script** and **Script_Extensions**
<table>
<thead>
<tr>
<th>Font</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cypriot</td>
<td></td>
</tr>
<tr>
<td>Cprt</td>
<td></td>
</tr>
<tr>
<td>Cyrillic</td>
<td></td>
</tr>
<tr>
<td>Cyrl</td>
<td></td>
</tr>
<tr>
<td>Deseret</td>
<td></td>
</tr>
<tr>
<td>Dsrt</td>
<td></td>
</tr>
<tr>
<td>Devanagari</td>
<td></td>
</tr>
<tr>
<td>Deva</td>
<td></td>
</tr>
<tr>
<td>Dives_Akuru</td>
<td></td>
</tr>
<tr>
<td>Diak</td>
<td></td>
</tr>
<tr>
<td>Dogra</td>
<td></td>
</tr>
<tr>
<td>Dogr</td>
<td></td>
</tr>
<tr>
<td>Duployan</td>
<td></td>
</tr>
<tr>
<td>Dupl</td>
<td></td>
</tr>
<tr>
<td>Egyptian_Hieroglyphs</td>
<td></td>
</tr>
<tr>
<td>Egyp</td>
<td></td>
</tr>
<tr>
<td>Elbasan</td>
<td></td>
</tr>
<tr>
<td>Elba</td>
<td></td>
</tr>
<tr>
<td>Elymaic</td>
<td></td>
</tr>
<tr>
<td>Elym</td>
<td></td>
</tr>
<tr>
<td>Ethiopic</td>
<td></td>
</tr>
<tr>
<td>Ethi</td>
<td></td>
</tr>
<tr>
<td>Georgian</td>
<td></td>
</tr>
<tr>
<td>Geor</td>
<td></td>
</tr>
<tr>
<td>Glagolitic</td>
<td></td>
</tr>
<tr>
<td>Glag</td>
<td></td>
</tr>
<tr>
<td>Gothic</td>
<td></td>
</tr>
<tr>
<td>Goth</td>
<td></td>
</tr>
<tr>
<td>Grantha</td>
<td></td>
</tr>
<tr>
<td>Gran</td>
<td></td>
</tr>
<tr>
<td>Greek</td>
<td></td>
</tr>
<tr>
<td>Grek</td>
<td></td>
</tr>
<tr>
<td>Gujarati</td>
<td></td>
</tr>
<tr>
<td>Gujr</td>
<td></td>
</tr>
<tr>
<td>Gunjala_Gondi</td>
<td></td>
</tr>
<tr>
<td>Gong</td>
<td></td>
</tr>
<tr>
<td>Script</td>
<td>Script</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Gurmukhi</td>
<td>Gurmukhi</td>
</tr>
<tr>
<td>Guru</td>
<td></td>
</tr>
<tr>
<td>Han</td>
<td>Han</td>
</tr>
<tr>
<td>Hani</td>
<td></td>
</tr>
<tr>
<td>Hangul</td>
<td>Hangul</td>
</tr>
<tr>
<td>Hang</td>
<td></td>
</tr>
<tr>
<td>Hanifi_Rohingya</td>
<td>Hanifi_Rohingya</td>
</tr>
<tr>
<td>Rohg</td>
<td></td>
</tr>
<tr>
<td>Hanunoo</td>
<td>Hanunoo</td>
</tr>
<tr>
<td>Hano</td>
<td></td>
</tr>
<tr>
<td>Hatran</td>
<td>Hatran</td>
</tr>
<tr>
<td>Hatr</td>
<td></td>
</tr>
<tr>
<td>Hebrew</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Hebr</td>
<td></td>
</tr>
<tr>
<td>Hiragana</td>
<td>Hiragana</td>
</tr>
<tr>
<td>Hira</td>
<td></td>
</tr>
<tr>
<td>Imperial_Aramaic</td>
<td>Imperial_Aramaic</td>
</tr>
<tr>
<td>Armi</td>
<td></td>
</tr>
<tr>
<td>Inherited</td>
<td>Inherited</td>
</tr>
<tr>
<td>Zinh</td>
<td></td>
</tr>
<tr>
<td>Qaai</td>
<td></td>
</tr>
<tr>
<td>Inscriptional_Pahlavi</td>
<td>Inscriptional_Pahlavi</td>
</tr>
<tr>
<td>Phli</td>
<td></td>
</tr>
<tr>
<td>Inscriptional_Parthian</td>
<td>Inscriptional_Parthian</td>
</tr>
<tr>
<td>Prti</td>
<td></td>
</tr>
<tr>
<td>Javanese</td>
<td>Javanese</td>
</tr>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>Kaithi</td>
<td>Kaithi</td>
</tr>
<tr>
<td>Kthi</td>
<td></td>
</tr>
<tr>
<td>Kannada</td>
<td>Kannada</td>
</tr>
<tr>
<td>Knda</td>
<td></td>
</tr>
<tr>
<td>Katakana</td>
<td>Katakana</td>
</tr>
<tr>
<td>Kana</td>
<td></td>
</tr>
<tr>
<td>Kayah_Li</td>
<td>Kayah_Li</td>
</tr>
<tr>
<td>Kali</td>
<td></td>
</tr>
<tr>
<td>Kharoshthi</td>
<td>Kharoshthi</td>
</tr>
<tr>
<td>Script</td>
<td>Script</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Khar</td>
<td>Khitan_Small_Script</td>
</tr>
<tr>
<td>Kits</td>
<td>Khitan_Small_Script</td>
</tr>
<tr>
<td>Khmer</td>
<td>Khmer</td>
</tr>
<tr>
<td>Khmr</td>
<td>Khojki</td>
</tr>
<tr>
<td>Khojki</td>
<td>Khudawadi</td>
</tr>
<tr>
<td>Khoj</td>
<td>Lao</td>
</tr>
<tr>
<td>Khmr</td>
<td>Laoo</td>
</tr>
<tr>
<td>Lao</td>
<td>Latin</td>
</tr>
<tr>
<td>Laoo</td>
<td>Latin</td>
</tr>
<tr>
<td>Lao</td>
<td>Lepcha</td>
</tr>
<tr>
<td>Laoo</td>
<td>Lepcha</td>
</tr>
<tr>
<td>Lepcha</td>
<td>Limbu</td>
</tr>
<tr>
<td>Lepc</td>
<td>Limbu</td>
</tr>
<tr>
<td>Limbu</td>
<td>Linear_A</td>
</tr>
<tr>
<td>Limb</td>
<td>Linear_A</td>
</tr>
<tr>
<td>Linear_A</td>
<td>Lina</td>
</tr>
<tr>
<td>Linear_B</td>
<td>Lina</td>
</tr>
<tr>
<td>Lina</td>
<td>Linb</td>
</tr>
<tr>
<td>Linb</td>
<td>Lisu</td>
</tr>
<tr>
<td>Lisu</td>
<td>Lycian</td>
</tr>
<tr>
<td>Lycian</td>
<td>Lycian</td>
</tr>
<tr>
<td>Lyci</td>
<td>Lydian</td>
</tr>
<tr>
<td>Lydian</td>
<td>Lydian</td>
</tr>
<tr>
<td>Lydi</td>
<td>Mahajani</td>
</tr>
<tr>
<td>Mahajani</td>
<td>Mahajani</td>
</tr>
<tr>
<td>Mahj</td>
<td>Makasar</td>
</tr>
<tr>
<td>Makasar</td>
<td>Makasar</td>
</tr>
<tr>
<td>Maka</td>
<td>Malayalam</td>
</tr>
<tr>
<td>Malayalam</td>
<td>Malayalam</td>
</tr>
<tr>
<td>Mlym</td>
<td>Mandaic</td>
</tr>
<tr>
<td>Mandaic</td>
<td>Mand</td>
</tr>
<tr>
<td>Mand</td>
<td>Manichaean</td>
</tr>
<tr>
<td>Manichaean</td>
<td>Manichaean</td>
</tr>
<tr>
<td>Mani</td>
<td></td>
</tr>
<tr>
<td></td>
<td>607</td>
</tr>
<tr>
<td>Marchen</td>
<td>Marchen</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Marc</td>
<td></td>
</tr>
<tr>
<td>Medefaidrin</td>
<td>Medefaidrin</td>
</tr>
<tr>
<td>Medf</td>
<td></td>
</tr>
<tr>
<td>Masaram_Gondi</td>
<td>Masaram_Gondi</td>
</tr>
<tr>
<td>Gonm</td>
<td></td>
</tr>
<tr>
<td>Meetei_Mayek</td>
<td>Meetei_Mayek</td>
</tr>
<tr>
<td>Mtei</td>
<td></td>
</tr>
<tr>
<td>Mende_Kikakui</td>
<td>Mende_Kikakui</td>
</tr>
<tr>
<td>Mend</td>
<td></td>
</tr>
<tr>
<td>Meroitic_Cursive</td>
<td>Meroitic_Cursive</td>
</tr>
<tr>
<td>Merc</td>
<td></td>
</tr>
<tr>
<td>Meroitic_Hieroglyphs</td>
<td>Meroitic_Hieroglyphs</td>
</tr>
<tr>
<td>Mero</td>
<td></td>
</tr>
<tr>
<td>Miao</td>
<td></td>
</tr>
<tr>
<td>Plrd</td>
<td></td>
</tr>
<tr>
<td>Modi</td>
<td></td>
</tr>
<tr>
<td>Mongolian</td>
<td>Mongolian</td>
</tr>
<tr>
<td>Mong</td>
<td></td>
</tr>
<tr>
<td>Mro</td>
<td></td>
</tr>
<tr>
<td>Mroo</td>
<td></td>
</tr>
<tr>
<td>Multani</td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
</tr>
<tr>
<td>Myanmar</td>
<td>Myanmar</td>
</tr>
<tr>
<td>Mymr</td>
<td></td>
</tr>
<tr>
<td>Nabataean</td>
<td>Nabataean</td>
</tr>
<tr>
<td>Nbat</td>
<td></td>
</tr>
<tr>
<td>Nandinagari</td>
<td>Nandinagari</td>
</tr>
<tr>
<td>Nand</td>
<td></td>
</tr>
<tr>
<td>New_Tai_Lue</td>
<td>New_Tai_Lue</td>
</tr>
<tr>
<td>Talu</td>
<td></td>
</tr>
<tr>
<td>Newa</td>
<td></td>
</tr>
<tr>
<td>Nko</td>
<td></td>
</tr>
<tr>
<td>Nkoo</td>
<td></td>
</tr>
<tr>
<td>Nkoo</td>
<td></td>
</tr>
<tr>
<td>Nushu</td>
<td></td>
</tr>
<tr>
<td>Nshu</td>
<td></td>
</tr>
<tr>
<td>Nyiakeng_Puachue_Hmong</td>
<td>Nyiakeng_Puachue_Hmong</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Hmp</td>
<td></td>
</tr>
<tr>
<td>Ogham</td>
<td>Ogham</td>
</tr>
<tr>
<td>Ogam</td>
<td></td>
</tr>
<tr>
<td>Ol_Chiki</td>
<td>Ol_Chiki</td>
</tr>
<tr>
<td>Olck</td>
<td></td>
</tr>
<tr>
<td>Old_Hungarian</td>
<td>Old_Hungarian</td>
</tr>
<tr>
<td>Hung</td>
<td></td>
</tr>
<tr>
<td>Old_Italic</td>
<td>Old_Italic</td>
</tr>
<tr>
<td>Ital</td>
<td></td>
</tr>
<tr>
<td>Old_North_Arabian</td>
<td>Old_North_Arabian</td>
</tr>
<tr>
<td>Narb</td>
<td></td>
</tr>
<tr>
<td>Old_Permic</td>
<td>Old_Permic</td>
</tr>
<tr>
<td>Perm</td>
<td></td>
</tr>
<tr>
<td>Old_Persian</td>
<td>Old_Persian</td>
</tr>
<tr>
<td>Xpeo</td>
<td></td>
</tr>
<tr>
<td>Old_Sogdian</td>
<td>Old_Sogdian</td>
</tr>
<tr>
<td>Sogo</td>
<td></td>
</tr>
<tr>
<td>Old_South_Arabian</td>
<td>Old_South_Arabian</td>
</tr>
<tr>
<td>Sarb</td>
<td></td>
</tr>
<tr>
<td>Old_Turkic</td>
<td>Old_Turkic</td>
</tr>
<tr>
<td>Orkh</td>
<td></td>
</tr>
<tr>
<td>Oriya</td>
<td>Oriya</td>
</tr>
<tr>
<td>Orya</td>
<td></td>
</tr>
<tr>
<td>Osage</td>
<td>Osage</td>
</tr>
<tr>
<td>Osge</td>
<td></td>
</tr>
<tr>
<td>Osmanyia</td>
<td>Osmanyia</td>
</tr>
<tr>
<td>Osma</td>
<td></td>
</tr>
<tr>
<td>Phahw_Hmong</td>
<td>Phahw_Hmong</td>
</tr>
<tr>
<td>Hmng</td>
<td></td>
</tr>
<tr>
<td>Palmyrene</td>
<td>Palmyrene</td>
</tr>
<tr>
<td>Palm</td>
<td></td>
</tr>
<tr>
<td>Pau_Cin_Hau</td>
<td>Pau_Cin_Hau</td>
</tr>
<tr>
<td>Pauc</td>
<td></td>
</tr>
<tr>
<td>Phags_Pa</td>
<td>Phags_Pa</td>
</tr>
<tr>
<td>Phag</td>
<td></td>
</tr>
<tr>
<td>Phoenician</td>
<td>Phoenician</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>Phnx</td>
<td>Phnx</td>
</tr>
<tr>
<td>Psalter_Pahlavi</td>
<td>Psalter_Pahlavi</td>
</tr>
<tr>
<td>Phlp</td>
<td>Rejang</td>
</tr>
<tr>
<td>Rejang</td>
<td>Rjng</td>
</tr>
<tr>
<td>Runic</td>
<td>Runr</td>
</tr>
<tr>
<td>Samaritan</td>
<td>Samaritan</td>
</tr>
<tr>
<td>Samr</td>
<td>Saurashtra</td>
</tr>
<tr>
<td>Saur</td>
<td>Saur</td>
</tr>
<tr>
<td>Sharada</td>
<td>Sharada</td>
</tr>
<tr>
<td>Shrd</td>
<td>Shrd</td>
</tr>
<tr>
<td>Shavian</td>
<td>Shavian</td>
</tr>
<tr>
<td>Shaw</td>
<td>Siddham</td>
</tr>
<tr>
<td>Sidd</td>
<td>SignWriting</td>
</tr>
<tr>
<td>Sgnw</td>
<td>Sinhala</td>
</tr>
<tr>
<td>Sinh</td>
<td>Sogdian</td>
</tr>
<tr>
<td>Sogd</td>
<td>Sogd</td>
</tr>
<tr>
<td>Sora_Sompeng</td>
<td>Sora_Sompeng</td>
</tr>
<tr>
<td>Sora</td>
<td>Soyombo</td>
</tr>
<tr>
<td>Soyombo</td>
<td>Soyombo</td>
</tr>
<tr>
<td>Soyo</td>
<td>Sundanese</td>
</tr>
<tr>
<td>Sundanese</td>
<td>Sundanese</td>
</tr>
<tr>
<td>Sund</td>
<td>Syloti_Nagri</td>
</tr>
<tr>
<td>Syloti_Nagri</td>
<td>Sylo</td>
</tr>
<tr>
<td>Sylo</td>
<td>Syriac</td>
</tr>
<tr>
<td>Syriac</td>
<td>Syrc</td>
</tr>
<tr>
<td>Syrc</td>
<td>Tagalog</td>
</tr>
<tr>
<td>Tagalog</td>
<td>Tglg</td>
</tr>
<tr>
<td>Tglg</td>
<td></td>
</tr>
<tr>
<td>Language Name</td>
<td>Language Name</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Tagbanwa</td>
<td>Tagbanwa</td>
</tr>
<tr>
<td>Tagb</td>
<td>Tagb</td>
</tr>
<tr>
<td>Tai_Le</td>
<td>Tai_Le</td>
</tr>
<tr>
<td>Tale</td>
<td>Tale</td>
</tr>
<tr>
<td>Tai_Tham</td>
<td>Tai_Tham</td>
</tr>
<tr>
<td>Lana</td>
<td>Lana</td>
</tr>
<tr>
<td>Tai_Viet</td>
<td>Tai_Viet</td>
</tr>
<tr>
<td>Tavt</td>
<td>Tavt</td>
</tr>
<tr>
<td>Takri</td>
<td>Takri</td>
</tr>
<tr>
<td>Takr</td>
<td>Takr</td>
</tr>
<tr>
<td>Tamil</td>
<td>Tamil</td>
</tr>
<tr>
<td>Taml</td>
<td>Taml</td>
</tr>
<tr>
<td>Tangut</td>
<td>Tangut</td>
</tr>
<tr>
<td>Tang</td>
<td>Tang</td>
</tr>
<tr>
<td>Telugu</td>
<td>Telugu</td>
</tr>
<tr>
<td>Telu</td>
<td>Telu</td>
</tr>
<tr>
<td>Thaana</td>
<td>Thaana</td>
</tr>
<tr>
<td>Thaa</td>
<td>Thaa</td>
</tr>
<tr>
<td>Thai</td>
<td>Thai</td>
</tr>
<tr>
<td>Tibetan</td>
<td>Tibetan</td>
</tr>
<tr>
<td>Tibt</td>
<td>Tibt</td>
</tr>
<tr>
<td>Tifinagh</td>
<td>Tifinagh</td>
</tr>
<tr>
<td>Tfng</td>
<td>Tfng</td>
</tr>
<tr>
<td>Tirhutha</td>
<td>Tirhutha</td>
</tr>
<tr>
<td>Tirh</td>
<td>Tirh</td>
</tr>
<tr>
<td>Ugaritic</td>
<td>Ugaritic</td>
</tr>
<tr>
<td>Ugar</td>
<td>Ugar</td>
</tr>
<tr>
<td>Vai</td>
<td>Vai</td>
</tr>
<tr>
<td>Vaii</td>
<td>Vaii</td>
</tr>
<tr>
<td>Wancho</td>
<td>Wancho</td>
</tr>
<tr>
<td>Wcho</td>
<td>Wcho</td>
</tr>
<tr>
<td>Warang_Citi</td>
<td>Warang_Citi</td>
</tr>
<tr>
<td>Wara</td>
<td>Wara</td>
</tr>
<tr>
<td>Yezidi</td>
<td>Yezidi</td>
</tr>
<tr>
<td>Yezi</td>
<td>Yezi</td>
</tr>
<tr>
<td>Yi</td>
<td>Yi</td>
</tr>
</tbody>
</table>
22.2.9 AtomEscape

With parameter \textit{direction}.

The production \textit{AtomEscape} :: \textit{DecimalEscape} evaluates as follows:

1. Evaluate \textit{DecimalEscape} to obtain an integer \textit{n}.
2. Assert: \textit{n} \leq \textit{NcapturingParens}.
3. Return ! BackreferenceMatcher(\textit{n}, \textit{direction}).

The production \textit{AtomEscape} :: \textit{CharacterEscape} evaluates as follows:

1. Evaluate \textit{CharacterEscape} to obtain a character \textit{ch}.
2. Let \textit{A} be a one-element CharSet containing the character \textit{ch}.
3. Return ! CharacterSetMatcher(\textit{A}, \textit{false}, \textit{direction}).

The production \textit{AtomEscape} :: \textit{CharacterClassEscape} evaluates as follows:

1. Evaluate \textit{CharacterClassEscape} to obtain a CharSet \textit{A}.
2. Return ! CharacterSetMatcher(\textit{A}, \textit{false}, \textit{direction}).

\textbf{NOTE}

An escape sequence of the form $\backslash$ followed by a non-zero decimal number \textit{n} matches the result of the \textit{n}th set of capturing parentheses (22.2.2.1). It is an error if the regular expression has fewer than \textit{n} capturing parentheses. If the regular expression has \textit{n} or more capturing parentheses but the \textit{n}th one is \textit{undefined} because it has not captured anything, then the backreference always succeeds.

The production \textit{AtomEscape} :: \textit{k} \textit{GroupName} evaluates as follows:

1. Search the enclosing \textit{Pattern} for an instance of a \textit{GroupSpecifier} containing a \textit{RegExpIdentifierName} which has a \textit{CapturingGroupName} equal to the \textit{CapturingGroupName} of the \textit{RegExpIdentifierName} contained in \textit{GroupName}.
2. Assert: A unique such \textit{GroupSpecifier} is found.
3. Let \textit{parenIndex} be the number of left-capturing parentheses in the entire regular expression that occur to the left of the located \textit{GroupSpecifier}. This is the total number of \textit{Atom} :: ( \textit{GroupSpecifier} \textit{Disjunction} ) Parse Nodes prior to or enclosing the located \textit{GroupSpecifier}, including its immediately enclosing \textit{Atom}.
4. Return ! BackreferenceMatcher(\textit{parenIndex}, \textit{direction}).

22.2.9.1 BackreferenceMatcher ( \textit{n}, \textit{direction} )

The abstract operation BackreferenceMatcher takes arguments \textit{n} (a positive integer) and \textit{direction} (1 or -1). It performs the following steps when called:

1. Assert: \textit{n} \geq 1.
2. Return a new Matcher with parameters \textit{(x, c)} that captures \textit{n} and \textit{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 \( e + \text{direction} \times \text{len} \).
  i. If \( f < 0 \) or \( f > \text{InputLength} \), return failure.
  j. Let \( g \) be \( \text{min}(e, f) \).
  k. If there exists an integer \( i \) between 0 (inclusive) and \( len \) (exclusive) such that \( \text{Canonicalize}(s[i]) \) is not the same character value as \( \text{Canonicalize(Input}[g + i]) \), 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:

\[
\text{CharacterEscape} \::= \\
\text{ControlEscape} \\
\text{ControlLetter} \\
\text{0 [lookahead} \notin \text{DecimalDigit]} \\
\text{HexEscapeSequence} \\
\text{RegExpUnicodeEscapeSequence} \\
\text{IdentityEscape}
\]

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:

\[
\text{DecimalEscape} \::= \text{NonZeroDigit} \text{ DecimalDigit}^* \text{ opt}
\]

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 \( \text{CharacterClassEscape} \::= \text{d} \) evaluates as follows:

1. Return the ten-element CharSet containing the characters \( 0 \) through \( 9 \) inclusive.
The production \( \text{CharacterClassEscape} :: \, \mathbf{d} \) evaluates as follows:

1. Return the CharSet containing all characters not in the CharSet returned by \( \text{CharacterClassEscape} :: \, \mathbf{d} \).

The production \( \text{CharacterClassEscape} :: \, \mathbf{s} \) evaluates as follows:

1. Return the CharSet containing all characters corresponding to a code point on the right-hand side of the \( \text{WhiteSpace} \) or \( \text{LineTerminator} \) productions.

The production \( \text{CharacterClassEscape} :: \, \mathbf{w} \) evaluates as follows:

1. Return \( \text{WordCharacters} \).

The production \( \text{CharacterClassEscape} :: \, \mathbf{W} \) evaluates as follows:

1. Return the CharSet containing all characters not in the CharSet returned by \( \text{CharacterClassEscape} :: \, \mathbf{w} \).

The production \( \text{CharacterClassEscape} :: \, \mathbf{p}\{ \text{UnicodePropertyValueExpression} \} \) evaluates as follows:

1. Return the CharSet containing all Unicode code points included in the CharSet returned by \( \text{UnicodePropertyValueExpression} \).

The production \( \text{CharacterClassEscape} :: \, \mathbf{P}\{ \text{UnicodePropertyValueExpression} \} \) evaluates as follows:

1. Return the CharSet containing all Unicode code points not included in the CharSet returned by \( \text{UnicodePropertyValueExpression} \).

The production \( \text{UnicodePropertyValueExpression} :: \, \text{UnicodePropertyName} = \text{UnicodePropertyValue} \) evaluates as follows:

1. Let \( ps \) be SourceText of \( \text{UnicodePropertyName} \).
2. Let \( p \) be ! \( \text{UnicodeMatchProperty}(ps) \).
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 \( \text{UnicodePropertyValue} \).
5. Let \( v \) be ! \( \text{UnicodeMatchPropertyValue}(p, vs) \).
6. Return the CharSet containing all Unicode code points whose character database definition includes the property \( p \) with value \( v \).

The production \( \text{UnicodePropertyValueExpression} :: \, \text{LoneUnicodePropertyNameOrValue} \) evaluates as follows:

1. Let \( s \) be SourceText of \( \text{LoneUnicodePropertyNameOrValue} \).
2. If ! \( \text{UnicodeMatchPropertyValue}(\text{General_Category}, s) \) 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 ! \( \text{UnicodeMatchProperty}(s) \).
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 \( \text{CharacterClass} :: [ \text{ClassRanges} ] \) evaluates as follows:

1. Evaluate \( \text{ClassRanges} \) to obtain a CharSet \( A \).
2. Return the two results \( A \) and \( \text{false} \).

The production \( \text{CharacterClass} :: [ ^ \text{ClassRanges} ] \) evaluates as follows:

1. Evaluate \( \text{ClassRanges} \) to obtain a CharSet \( A \).
2. Return the two results \( A \) and \( \text{true} \).

22.2.2.14 ClassRanges

The production \( \text{ClassRanges} :: [\text{empty}] \) evaluates as follows:

1. Return the empty CharSet.

The production \( \text{ClassRanges} :: \text{NonemptyClassRanges} \) evaluates as follows:

1. Return the CharSet that is the result of evaluating \( \text{NonemptyClassRanges} \).

22.2.2.15 NonemptyClassRanges

The production \( \text{NonemptyClassRanges} :: \text{ClassAtom} \) evaluates as follows:

1. Return the CharSet that is the result of evaluating \( \text{ClassAtom} \).

The production \( \text{NonemptyClassRanges} :: \text{ClassAtom} \text{NonemptyClassRangesNoDash} \) evaluates as follows:

1. Evaluate \( \text{ClassAtom} \) to obtain a CharSet \( A \).
2. Evaluate \( \text{NonemptyClassRangesNoDash} \) to obtain a CharSet \( B \).
3. Return the union of CharSets \( A \) and \( B \).

The production \( \text{NonemptyClassRanges} :: \text{ClassAtom} - \text{ClassAtom} \text{ClassRanges} \) evaluates as follows:

1. Evaluate the first \( \text{ClassAtom} \) to obtain a CharSet \( A \).
2. Evaluate the second \( \text{ClassAtom} \) to obtain a CharSet \( B \).
3. Evaluate \( \text{ClassRanges} \) to obtain a CharSet \( C \).
4. Let \( D \) be \( ! \text{CharacterRange}(A, B) \).
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 \( \text{NonemptyClassRangesNoDash} \ ::= \text{ClassAtom} \) evaluates as follows:

1. Return the CharSet that is the result of evaluating \( \text{ClassAtom} \).

The production \( \text{NonemptyClassRangesNoDash} \ ::= \text{ClassAtomNoDash \ NonemptyClassRangesNoDash} \) evaluates as follows:

1. Evaluate \( \text{ClassAtomNoDash} \) to obtain a CharSet \( A \).
2. Evaluate \( \text{NonemptyClassRangesNoDash} \) to obtain a CharSet \( B \).
3. Return the union of CharSets \( A \) and \( B \).

The production \( \text{NonemptyClassRangesNoDash} \ ::= \text{ClassAtomNoDash} \ - \ \text{ClassAtom} \ \text{ClassRanges} \) evaluates as follows:

1. Evaluate \( \text{ClassAtomNoDash} \) to obtain a CharSet \( A \).
2. Evaluate \( \text{ClassAtom} \) to obtain a CharSet \( B \).
3. Evaluate \( \text{ClassRanges} \) to obtain a CharSet \( C \).
4. Let \( D \) be \( \text{CharacterRange}(A, B) \).
5. Return the union of \( D \) and \( C \).

**NOTE 1**  
\( \text{ClassRanges} \) can expand into a single \( \text{ClassAtom} \) and/or ranges of two \( \text{ClassAtom} \) separated by dashes. In the latter case the \( \text{ClassRanges} \) includes all characters between the first \( \text{ClassAtom} \) and the second \( \text{ClassAtom} \), inclusive; an error occurs if either \( \text{ClassAtom} \) does not represent a single character (for example, if one is \( \text{\w} \)) or if the first \( \text{ClassAtom} \)'s character value is greater than the second \( \text{ClassAtom} \)'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 \( \text{ClassRanges} \), the beginning or end limit of a range specification, or immediately follows a range specification.

**22.2.2.17 ClassAtom**

The production \( \text{ClassAtom} \ ::= \ - \) 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 \( \backslash \) or \( \) or - evaluates as follows:

1. Return the CharSet containing the character matched by \( SourceCharacter \).

The production \( ClassAtomNoDash :: \backslash ClassEscape \) evaluates as follows:

1. Return the CharSet that is the result of evaluating \( ClassEscape \).

### 22.2.19 ClassEscape

The \( ClassEscape \) productions evaluate as follows:

\[
\begin{align*}
ClassEscape :: & b \\
ClassEscape :: & - \\
ClassEscape :: & \text{CharacterEscape}
\end{align*}
\]

1. Let \( cv \) be the \text{CharacterValue} of this \( ClassEscape \).
2. Let \( c \) be the character whose character value is \( cv \).
3. Return the CharSet containing the single character \( c \).

\( ClassEscape :: \text{CharacterClassEscape} \)

1. Return the CharSet that is the result of evaluating \( \text{CharacterClassEscape} \).

### NOTE

A \( ClassAtom \) can use any of the escape sequences that are allowed in the rest of the regular expression except for \( \backslash b \), \( \backslash B \), and backreferences. Inside a \( \text{CharacterClass} \), \( \backslash b \) means the backspace character, while \( \backslash B \) and backreferences raise errors. Using a backreference inside a \( ClassAtom \) causes an error.

### 22.2.3 The RegExp Constructor

The RegExp \text{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 \text{constructor}. Thus the function call \text{RegExp(…)} is equivalent to the object creation expression \text{new RegExp(…)} with the same arguments.
- is designed to be subclassable. It may be used as the value of an \text{extends} clause of a class definition. Subclass constructors that intend to inherit the specified RegExp behaviour must include a \text{super} call to the RegExp \text{constructor} to create and initialize subclass instances with the necessary internal slots.
22.2.3.1 RegExp (pattern, flags)

The following steps are taken:

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.

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 \( \text{pattern} \) is \textit{undefined}, let \( \text{P} \) be the empty String.
2. Else, let \( \text{P} \) be \( \text{? ToString(\text{pattern})} \).
3. If \( \text{flags} \) is \textit{undefined}, let \( \text{F} \) be the empty String.
4. Else, let \( \text{F} \) be \( \text{? ToString(\text{flags})} \).
5. If \( \text{F} \) 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 \textit{SyntaxError} exception.
6. If \( \text{F} \) contains "u", let \( \text{u} \) be \textit{true}; else let \( \text{u} \) be \textit{false}.
7. If \( \text{u} \) is \textit{true}, then
   a. Let \( \text{patternText} \) be \( \text{! StringToCodePoints(\text{P})} \).
   b. Let \( \text{patternCharacters} \) be a List whose elements are the code points of \( \text{patternText} \).
8. Else,
   a. Let \( \text{patternText} \) be the result of interpreting each of \( \text{P} \)’s 16-bit elements as a Unicode BMP code point.
      UTF-16 decoding is not applied to the elements.
   b. Let \( \text{patternCharacters} \) be a List whose elements are the code unit elements of \( \text{P} \).
9. Let \( \text{parseResult} \) be \( \text{ParsePattern(\text{patternText}, \text{u})} \).
10. If \( \text{parseResult} \) is a non-empty List of \textit{SyntaxError} objects, throw a \textit{SyntaxError} exception.
11. \textbf{Assert:} \( \text{parseResult} \) is a Parse Node for \textit{Pattern}.
12. Set \( \text{obj}[:,:[\text{OriginalSource}]] \) to \( \text{P} \).
13. Set \( \text{obj}[:,:[\text{OriginalFlags}]] \) to \( \text{F} \).
14. Set \( \text{obj}[:,:[\text{RegExpMatcher}]] \) to the Abstract Closure that evaluates \( \text{parseResult} \) by applying the semantics provided in \ref{sec:22.2.2} using \( \text{patternCharacters} \) as the pattern’s List of \textit{SourceCharacter} values and \( \text{F} \) as the flag parameters.
15. Perform ? \text{Set(\text{obj}, "lastIndex", +0\textgreek{f}, true)}.
16. Return \( \text{obj} \).

\textbf{22.2.3.2.3 Static Semantics: ParsePattern ( \textit{patternText}, \textit{u} )}

The abstract operation \texttt{ParsePattern} takes arguments \( \text{patternText} \) (a sequence of Unicode code points) and \( \text{u} \) (a Boolean). It performs the following steps when called:

1. If \( \text{u} \) is \textit{true}, then
   a. Let \( \text{parseResult} \) be \( \texttt{ParseText(patternText, Pattern[+U, +N])} \).
2. Else,
   a. Let \( \text{parseResult} \) be \( \texttt{ParseText(patternText, Pattern[-U, -N])} \).
   b. If \( \text{parseResult} \) is a Parse Node and \( \text{parseResult} \) contains a GroupName, then
      i. Set \( \text{parseResult} \) to \( \texttt{ParseText(patternText, Pattern[-U, +N])} \).
3. Return \( \text{parseResult} \).

\textbf{22.2.3.2.4 RegExpCreate ( \textit{P}, \textit{F} )}

The abstract operation \texttt{RegExpCreate} takes arguments \( \text{P} \) and \( \text{F} \). It performs the following steps when called:

1. Let \( \text{obj} \) be ? \texttt{RegExpAlloc(%RegExp%)}.
2. Return ? \texttt{RegExpInitialize(obj, P, F)}.

\textbf{22.2.3.2.5 EscapeRegExpPattern ( \textit{P}, \textit{F} )}

The abstract operation \texttt{EscapeRegExpPattern} takes arguments \( \text{P} \) and \( \text{F} \). 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.
3. Let `S` be ? `ToString(string)`.

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
   b. If `Type(result)` is neither Object nor Null, throw a `TypeError` exception.
   c. Return `result`.

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 `ℝ(? ToLength(? Get(R, "lastIndex")))`. 
5. Let $flags$ be $R.\{[\text{OriginalFlags}]\}$.
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.
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 $lastIndex > \text{length}$, then
      i. If $global$ is true or $sticky$ is true, then
         1. Perform $\text{Set}(R, \"lastIndex\", +0\|u1D53D, \true)$.
      ii. Return null.
   b. Let $r$ be $matcher(S, lastIndex)$.
   c. If $r$ is failure, then
      i. If $sticky$ is true, then
         1. Perform $\text{Set}(R, \"lastIndex\", +0\|u1D53D, \true)$.
         2. Return null.
      ii. Set $lastIndex$ to $\text{AdvanceStringIndex}(S, lastIndex, fullUnicode)$.
   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 $\text{Set}(R, \"lastIndex\", F(e), \true)$.
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", $F(lastIndex)$).
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).


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` ≤ 2^{53} - 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
6. Else,
a. **Assert:** `global` is `true`.
b. Let `fullUnicode` be `! ToBoolean(? Get(rx, "unicode"))`.
c. Perform `? Set(rx, "lastIndex", +0F, 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(/u1D53D(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%)`.
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`.

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", +0F, 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 \( \text{position} \) be \( \text{ToIntegerOrInfinity(} \text{Get(} \text{result}, \text{"index"})\text{)} \).

f. Set \( \text{position} \) to the result of clamping \( \text{position} \) between 0 and \( \text{length}\text{S} \).

g. Let \( \text{n} \) be 1.

h. Let \( \text{captures} \) be a new empty List.

i. Repeat, while \( \text{n} \leq \text{nCaptures} \),
   i. Let \( \text{capN} \) be \( \text{? Get(} \text{result}, \text{! ToString(} \text{u1D53D}(\text{n}))\text{)} \).
      ii. If \( \text{capN} \) is not \( \text{undefined} \), then
         1. Set \( \text{capN} \) to \( \text{? ToString(} \text{capN}\text{)} \).
      iii. Append \( \text{capN} \) as the last element of \( \text{captures} \).
         iv. Set \( \text{n} \) to \( \text{n} + 1 \).

j. Let \( \text{namedCaptures} \) be \( \text{? Get(} \text{result}, \text{"groups"}) \).

k. If \( \text{functionalReplace} \) is true, then
   i. Let \( \text{replacerArgs} \) be « \( \text{matched} \) ».
   ii. Append in List order the elements of \( \text{captures} \) to the end of the List \( \text{replacerArgs} \).
   iii. Append \( \text{u1D53D}(\text{position}) \) and \( \text{S} \) to \( \text{replacerArgs} \).
   iv. If \( \text{namedCaptures} \) is not \( \text{undefined} \), then
      1. Append \( \text{namedCaptures} \) as the last element of \( \text{replacerArgs} \).
   v. Let \( \text{replValue} \) be \( \text{? Call(} \text{replaceValue}, \text{undefined}, \text{replacerArgs}\text{)} \).
   vi. Let \( \text{replacement} \) be \( \text{? ToString(} \text{replValue}\text{)} \).

l. Else,
   i. If \( \text{namedCaptures} \) is not \( \text{undefined} \), then
      1. Set \( \text{namedCaptures} \) to \( \text{? ToObject(} \text{namedCaptures}\text{)} \).
   ii. Let \( \text{replacement} \) be \( \text{? GetSubstitution(} \text{matched}, \text{S}, \text{position}, \text{captures}, \text{namedCaptures}, \text{replaceValue}\text{)} \).

m. If \( \text{position} \geq \text{nextSourcePosition} \), then
   i. NOTE: \( \text{position} \) 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 \( \text{rx} \). In such cases, the corresponding substitution is ignored.
   ii. Set \( \text{accumulatedResult} \) to the string-concatenation of \( \text{accumulatedResult} \), the substring of \( \text{S} \) from \( \text{nextSourcePosition} \) to \( \text{position} \), and \( \text{replacement} \).
   iii. Set \( \text{nextSourcePosition} \) to \( \text{position} + \text{matchLength} \).

15. If \( \text{nextSourcePosition} \geq \text{length}\text{S} \), return \( \text{accumulatedResult} \).

16. Return the string-concatenation of \( \text{accumulatedResult} \) and the substring of \( \text{S} \) from \( \text{nextSourcePosition} \).

The value of the "name" property of this function is "\[\text{Symbol.replace}\]".

### 22.2.5.11 RegExp.prototype [ @@search ] ( string )

When the \( \text{@@search} \) method is called with argument \( \text{string} \), the following steps are taken:

1. Let \( \text{rx} \) be the \text{this} value.
2. If \( \text{Type(} \text{rx}\text{)} \) is not Object, throw a \text{TypeError} exception.
3. Let \( \text{S} \) be \( \text{? ToString(} \text{string}\text{)} \).
4. Let \( \text{previousLastIndex} \) be \( \text{? Get(} \text{rx}, \text{"lastIndex"}\text{)} \).
5. If \( \text{SameValue(} \text{previousLastIndex}, +0\text{)} \) is false, then
   a. Perform \( \text{? Set(} \text{rx}, \text{"lastIndex"}, +0\text{, } \text{true}\text{)} \).
6. Let \( \text{result} \) be \( \text{? RegExpExec(} \text{rx}, \text{S}\text{)} \).
7. Let \( \text{currentLastIndex} \) be \( \text{? Get(} \text{rx}, \text{"lastIndex"}\text{)} \).
8. If \( \text{SameValue(} \text{currentLastIndex}, \text{previousLastIndex}\text{)} \) is false, then
a. Perform \( \text{Set}(rx, "lastIndex", previousLastIndex, true) \).

9. If \( result \) is \text{null}, return -1.

10. Return \( \text{Get}(result, "index") \).

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 \text{this} value.
2. If \( \text{Type}(R) \) is not \text{Object}, throw a \text{TypeError} exception.
3. If \( R \) does not have an \[[\text{OriginalSource}]\] internal slot, then
   a. If \text{SameValue}(R, %RegExp.prototype%) is \text{true}, return "(?:)".
   b. Otherwise, throw a \text{TypeError} exception.
4. Assert: \( R \) has an \[[\text{OriginalFlags}]\] internal slot.
5. Let \( src \) be \( R.[[\text{OriginalSource}]] \).
6. Let \( flags \) be \( R.[[\text{OriginalFlags}]] \).
7. Return \( \text{EscapeRegExpPattern}(src, flags) \).

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 `[", "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 `ℝ(? ToUint32(limit))`.
14. If `lim` is 0, return `A`.
15. Let `size` be the length of `S`.

629
16. If size is 0, then
   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 F(? 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).
         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).
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,
      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 ℝ(? ToLength(? Get(R, "lastIndex"))).
          2. Let nextIndex be ! AdvanceStringIndex(S, thisIndex, fullUnicode).
          3. Perform ? Set(R, "lastIndex", 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 "Reg Exp 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(\(\ldots\)) is equivalent to the object creation expression new Array(\(\ldots\)) 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 1.

23.1.1.1  Array (\(\ldots values\))

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`.
   Else,
      i. Let `intLen` be `! ToUint32(len)`.
      ii. If `intLen` is not the same value as `len`, throw a `RangeError` exception.
   d. 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)`. 

634
c. Let iteratorRecord be \( \text{GetIterator}(\text{items}, \text{sync}, \text{usingIterator}) \).
d. Let \( k \) be 0.
e. Repeat,
   i. If \( k \geq 2^{53} - 1 \), then
      1. Let error be \( \text{ThrowCompletion}(\text{a newly created TypeError object}) \).
      2. Return \( \text{IteratorClose}(\text{iteratorRecord, error}) \).
   ii. Let \( Pk \) be \( \text{ToString}(f(k)) \).
   iii. Let next be \( \text{IteratorStep}(\text{iteratorRecord}) \).
   iv. If next is false, then
      1. Perform \( \text{Set}(A, \"length", f(k), \text{true}) \).
      2. Return \( A \).
   v. Let nextValue be \( \text{IteratorValue}(\text{next}) \).
   vi. If mapping is true, then
      1. Let mappedValue be \( \text{Call}(\text{mapfn, thisArg, \{nextValue, f(k)} \}) \).
      2. If mappedValue is an abrupt completion, return \( \text{IteratorClose}(\text{iteratorRecord, mappedValue}) \).
      3. Set mappedValue to \( \text{mappedValue}[[\text{Value}}] \).
   vii. Else, let mappedValue be nextValue.
   viii. Let defineStatus be \( \text{CreateDataPropertyOrThrow}(A, Pk, \text{mappedValue}) \).
   ix. If defineStatus is an abrupt completion, return \( \text{IteratorClose}(\text{iteratorRecord, defineStatus}) \).
   x. Set \( k \) to \( k + 1 \).
6. NOTE: \text{items} is not an Iteratable so assume it is an array-like object.
7. Let arrayLike be \( \text{ToObject}(\text{items}) \).
8. Let len be \( \text{LengthOfArrayLike}(\text{arrayLike}) \).
9. If \( \text{IsConstructor}(C) \) is true, then
   a. Let \( A \) be \( \text{Construct}(C, \{f(len) \}) \).
10. Else,
    a. Let \( A \) be \( \text{ArrayCreate}(\text{len}) \).
11. Let \( k \) be 0.
12. Repeat, while \( k < \text{len} \),
   a. Let \( Pk \) be \( \text{ToString}(f(k)) \).
   b. Let \( kValue \) be \( \text{Get(arrayLike, Pk)} \).
   c. If mapping is true, then
      i. Let mappedValue be \( \text{Call}(\text{mapfn, thisArg, \{kValue, f(k)} \}) \).
   d. Else, let mappedValue be kValue.
   e. Perform \( \text{CreateDataPropertyOrThrow}(A, Pk, \text{mappedValue}) \).
   f. Set \( k \) to \( k + 1 \).
13. Perform \( \text{Set}(A, \"length", f(len), \text{true}) \).
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 \( \text{arg} \), 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)`.

**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_1D53D` and whose attributes are `{{Writable]: true, [Enumerable]: false, [[Configurable]]: false}_{1D53D}`. 
- 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 ≥ 2^{53} - 1`, throw a **TypeError** exception. 
      iii. Perform `CreateDataPropertyOrThrow(A, !ToString(F(n)), E)`. 
   6. Perform `Set(A, "length", F(n), true)`. 
   7. Return `A`. 

The "length" property of the **concat** method is `1_1D53D`.
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`.

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 `-∞`, 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 `-∞`, 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 `-∞`, 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`.

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.

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.

### 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`.

### 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.
c. Set to to to + count - 1.

17. Else,
a. Let direction be 1.

18. Repeat, while count > 0,
a. Let fromKey be ! ToString(F(from)).
b. Let toKey be ! ToString(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`. 
The following steps are taken:

1. Let \( O \) be ? ToObject\( (this) \).
2. Let \( \text{len} \) be ? LengthOfArrayLike\( (O) \).
3. Let \( \text{relativeStart} \) be ? ToIntegerOrInfinity\( (\text{start}) \).
4. If \( \text{relativeStart} \) is \(-\infty\), let \( k \) be 0.
5. Else if \( \text{relativeStart} < 0 \), let \( k \) be \( \max(\text{len} + \text{relativeStart}, 0) \).
6. Else, let \( k \) be \( \min(\text{relativeStart}, \text{len}) \).
7. If \( \text{end} \) is \( \text{undefined} \), let \( \text{relativeEnd} \) be \( \text{len} \); else let \( \text{relativeEnd} \) be ? ToIntegerOrInfinity\( (\text{end}) \).
8. If \( \text{relativeEnd} \) is \(-\infty\), let \( \text{final} \) be 0.
9. Else if \( \text{relativeEnd} < 0 \), let \( \text{final} \) be \( \max(\text{len} + \text{relativeEnd}, 0) \).
10. Else, let \( \text{final} \) be \( \min(\text{relativeEnd}, \text{len}) \).
11. Repeat, while \( k < \text{final} \),
   a. Let \( P_k \) be ? ToString\( (\mathbb{F}(k)) \).
   b. Perform ? Set\( (O, P_k, \text{value}, \text{true}) \).
   c. Set \( k \) to \( k + 1 \).
12. Return \( O \).

The \text{fill} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

23.1.3.7 Array.prototype.filter ( \text{callbackfn} [, \text{thisArg} ] )

\text{callbackfn} should be a function that accepts three arguments and returns a value that is coercible to a Boolean value. \text{filter} calls \text{callbackfn} once for each element in the array, in ascending order, and constructs a new array of all the values for which \text{callbackfn} returns \text{true}. \text{callbackfn} is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a \text{thisArg} parameter is provided, it will be used as the \text{this} value for each invocation of \text{callbackfn}. If it is not provided, \text{undefined} is used instead.

\text{callbackfn} is called with three arguments: the value of the element, the index of the element, and the object being traversed.

\text{filter} does not directly mutate the object on which it is called but the object may be mutated by the calls to \text{callbackfn}.

The range of elements processed by \text{filter} is set before the first call to \text{callbackfn}. Elements which are appended to the array after the call to \text{filter} begins will not be visited by \text{callbackfn}. If existing elements of the array are changed their value as passed to \text{callbackfn} will be the value at the time \text{filter} visits them; elements that are deleted after the call to \text{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 $\text{ToObject(this value)}$.
2. Let $len$ be $\text{LengthOfArrayLike}(O)$.
3. If $\text{IsCallable(callbackfn)}$ is `false`, throw a `TypeError` exception.
4. Let $A$ be $\text{ArraySpeciesCreate}(O, 0)$.
5. Let $k$ be 0.
6. Let $to$ be 0.
7. Repeat, while $k < len$,
   a. Let $Pk$ be $\text{ToString}(F(k))$.
   b. Let $kPresent$ be $\text{HasProperty}(O, Pk)$.
   c. If $kPresent$ is `true`, then
      i. Let $kValue$ be $\text{Get}(O, Pk)$.
      ii. Let $selected$ be $\text{ToBoolean(\Call{callbackfn}{thisArg}{kValue}{k}{O})}$.
      iii. If $selected$ is `true`, then
          1. Perform $\text{CreateDataPropertyOrThrow}(A, \text{ToString}(F(to)), kValue)$.
          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 $k + 1$.

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).
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 $+0\infty$.
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 \( \text{CreateDataPropertyOrThrow}(target, \text{! ToString}(F(targetIndex)), element) \).
3. Set \( targetIndex \) to \( targetIndex + 1 \).
   d. Set \( sourceIndex \) to \( sourceIndex + 1 \).
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 \( \text{ToObject}(this \text{ value}) \).
2. Let \( sourceLen \) be \( \text{LengthOfArrayLike}(O) \).
3. If \( \text{! IsCallable}(mapperFunction) \) is false, throw a `TypeError` exception.
4. Let \( A \) be \( \text{ArraySpeciesCreate}(O, 0) \).
5. Perform \( \text{FlattenIntoArray}(A, O, sourceLen, 0, 1, mapperFunction, thisArg) \).
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 \( \text{ToObject}(this \text{ value}) \).
2. Let \( len \) be \( \text{LengthOfArrayLike}(O) \).
3. If \( \text{IsCallable}(callbackfn) \) is false, throw a `TypeError` exception.
4. Let \( k \) be 0.
5. Repeat, while \( k < len \),
   a. Let \( Pk \) be \( \text{! ToString}(F(k)) \).
   b. Let \( kPresent \) be \( \text{HasProperty}(O, Pk) \).
   c. If \( kPresent \) is true, then
      i. Let \( kValue \) be \( \text{Get}(O, Pk) \).
      ii. Perform \( \text{Call}(callbackfn, thisArg, \ll kValue, F(k), O \rr) \).
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 $+0E$ (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 $+0E$, it is used as the offset from the end of the array to compute `fromIndex`. If the computed index is less than $+0E$, 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` ≥ 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(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, -1F is returned.

The optional second argument *fromIndex* defaults to +0F (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1F is returned, i.e. the array will not be searched. If it is less than +0F, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than +0F, 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 -1F.
4. Let *n* be ? ToIntegerOrInfinity(*fromIndex*).
5. **Assert**: If *fromIndex* is undefined, then *n* is 0.
6. If *n* is +∞, return -1F.
7. Else if *n* is -∞, set *n* to 0.
8. If *n* ≥ 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(*k*)).
   b. If *kPresent* is true, then
      i. Let *elementK* be ? Get(*O*, ! ToString(*k*)).
      ii. Let *same* be the result of performing Strict Equality Comparison *searchElement* == *elementK*.
      iii. If *same* is true, return *k*.
   c. Set *k* to *k* + 1.
11. Return -1F.

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 $\text{ToString}(\text{separator})$.
5. Let $R$ be the empty String.
6. Let $k$ be 0.
7. Repeat, while $k < \text{len}$,
   a. If $k > 0$, set $R$ to the string-concatenation of $R$ and $sep$.
   b. Let $element$ be $\text{Get}(O, ! \text{ToString}(F(k)))$.
   c. If $element$ is $\text{undefined}$ or $\text{null}$, let $next$ be the empty String; otherwise, let $next$ be $\text{ToString}(element)$.
   d. Set $R$ to the string-concatenation of $R$ and $next$.
   e. Set $k$ to $k + 1$.
8. Return $R$.

NOTE 2
The $\text{join}$ function is intentionally generic; it does not require that its $\text{this}$ value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.

23.1.3.16 Array.prototype.keys ()

The following steps are taken:

1. Let $O$ be $\text{ToObject}(\text{this} \text{ value})$.
2. Return $\text{CreateArrayIterator}(O, \text{key})$.

23.1.3.17 Array.prototype.lastIndexOf (searchElement [, fromIndex ])

NOTE 1
$\text{lastIndexOf}$ compares $\text{searchElement}$ 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, $-1_F$ 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 $+0_F$, it is used as the offset from the end of the array to compute $fromIndex$. If the computed index is less than $+0_F$, $-1_F$ is returned.

When the $\text{lastIndexOf}$ method is called with one or two arguments, the following steps are taken:

1. Let $O$ be $\text{ToObject}(\text{this} \text{ value})$.
2. Let $len$ be $\text{LengthOfArrayLike}(O)$.
3. If $len$ is 0, return $-1_F$.
4. If $fromIndex$ is present, let $n$ be $\text{ToIntegerOrInfinity}(fromIndex)$; else let $n$ be $len - 1$.
5. If $n$ is $-\infty$, return $-1_F$.
6. If $n \geq 0$, then
   a. Let $k$ be $\text{min}(n, len - 1)$.
7. Else,
   a. Let $k$ be $len + n$.
8. Repeat, while $k \geq 0$,
   a. Let $kPresent$ be $\text{HasProperty}(O, ! \text{ToString}(F(k)))$.
   b. If $kPresent$ is true, then
      i. Let $elementK$ be $\text{Get}(O, ! \text{ToString}(F(k)))$.
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`.


**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(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, F(k), O »)`.
      iii. Perform `? CreateDataPropertyOrThrow(A, Pk, mappedValue)`.
   d. Set `k` to `k + 1`.

7. Return `A`.
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
   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).
   g. Return element.

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 $len + argCount > 2^{53} - 1$, 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 $len + 1$.
7. Return $F(len)$.

The "length" property of the push method is $1_F$.
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, Ï(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.
1. 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, 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))).
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)
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 -∞, 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 -∞, 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.
16. Return A.

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.

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

NOTE 2

NOTE 3
NOTE 1

When the method is called with one or two arguments, the following steps are taken:

1. Let \( O \) be \( \text{ToObject}(\text{this} \text{ value}) \).
2. Let \( \text{len} \) be \( \text{LengthOfArrayLike}(O) \).
3. If \( \text{IsCallable}(\text{callbackfn}) \) is \( \text{false} \), throw a TypeError exception.
4. Let \( k \) be 0.
5. Repeat, while \( k < \text{len} \),
   a. Let \( Pk \) be \( \text{ToString}(F(k)) \).
   b. Let \( k\text{Present} \) be \( \text{HasProperty}(O, Pk) \).
   c. If \( k\text{Present} \) is \( \text{true} \), then
      i. Let \( k\text{Value} \) be \( \text{Get}(O, Pk) \).
      ii. Let \( \text{testResult} \) be \( \text{ToBoolean}(\text{Call}(\text{callbackfn}, \text{thisArg}, \langle \text{kValue}, F(k), O \rangle)) \).
      iii. If \( \text{testResult} \) is \( \text{true} \), return \( \text{true} \).
   d. Set \( k \) to \( k + 1 \).
6. Return \( \text{false} \).

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(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(j), items[j], true).
    b. Set j to j + 1.
11. Repeat, while j < len,
    a. Perform ? DeletePropertyOrThrow(obj, ! ToString(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[π(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 π(j) < π(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 $a <_{\text{CF}} b$ means comparefn(a, b) < 0; $a =_{\text{CF}} b$ means comparefn(a, b) = 0 (of either sign); and $a >_{\text{CF}} b$ 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 `+0_F`.
2. If `x` is `undefined`, return `1_F`.
3. If `y` is `undefined`, return `-1_F`.
4. If `comparefn` is not `undefined`, then
   a. Let `v` be `? ToNumber(? Call(comparefn, undefined, « x, y »)).`
   b. If `v` is `NaN`, return `+0_F`.
   c. Return `v`.
5. Let `xString` be `? ToString(x)`.
7. Let `xSmaller` be the result of performing `Abstract Relational Comparison xString < yString`.
8. If `xSmaller` is `true`, return `-1_F`.
9. Let `ySmaller` be the result of performing `Abstract Relational Comparison yString < xString`.
10. If `ySmaller` is `true`, return `1_F`.
11. Return `+0_F`.

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)`
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 ∞, 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 > 2^{53} - 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. `  
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 \( \text{len} \).
d. Repeat, while \( k > (\text{len} - \text{actualDeleteCount} + \text{itemCount}) \),
   i. Perform \( ? \) \( \text{DeletePropertyOrThrow}(O, !\text{ToString}(F(k - 1))) \).
   ii. Set \( k \) to \( k - 1 \).

17. Else if \( \text{itemCount} > \text{actualDeleteCount} \), then
   a. Set \( k \) to \( (\text{len} - \text{actualDeleteCount}) \).
   b. Repeat, while \( k > \text{actualStart} \),
      i. Let \( \text{from} \) be \( !\text{ToString}(F(k + \text{actualDeleteCount} - 1)) \).
      ii. Let \( \text{to} \) be \( !\text{ToString}(F(k + \text{itemCount} - 1)) \).
      iii. Let \( \text{fromPresent} \) be \( ?\text{HasProperty}(O, \text{from}) \).
      iv. If \( \text{fromPresent} \) is \( \text{true} \), then
         1. Let \( \text{fromValue} \) be \( ?\text{Get}(O, \text{from}) \).
         2. Perform \( ?\text{Set}(O, \text{to}, \text{fromValue}, \text{true}) \).
      v. Else,
         1. Assert: \( \text{fromPresent} \) is \( \text{false} \).
         2. Perform \( ?\text{DeletePropertyOrThrow}(O, \text{to}) \).
   vi. Set \( k \) to \( k - 1 \).

18. Set \( k \) to \( \text{actualStart} \).
19. For each element \( E \) of \( \text{items} \), do
   a. Perform \( ?\text{Set}(O, !\text{ToString}(F(k)), E, \text{true}) \).
   b. Set \( k \) to \( k + 1 \).
20. Perform \( ?\text{Set}(O, "\text{length}", F(\text{len} - \text{actualDeleteCount} + \text{itemCount}), \text{true}) \).
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 ( [ \text{reserved1} [, \text{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($k - 1$).
      ii. Let $to$ be ! ToString($k + argCount - 1$).
      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 +0F.
   e. For each element $E$ of $items$, do
      i. Perform ? Set($O$, ! ToString($j$), $E$, true).
      ii. Set $j$ to $j + 1F$.
6. Return $len + argCount$.

The "length" property of the unshift method is 1F.

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).
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 ≥ 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

<table>
<thead>
<tr>
<th>Constructor Name and Intrinsic</th>
<th>Element Type</th>
<th>Element Size</th>
<th>Conversion Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Array %Int8Array%</td>
<td>Int8</td>
<td>1</td>
<td>ToInt8</td>
<td>8-bit two’s complement signed integer</td>
</tr>
<tr>
<td>Uint8Array %Uint8Array%</td>
<td>Uint8</td>
<td>1</td>
<td>ToUint8</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>Uint8ClampedArray %Uint8ClampedArray%</td>
<td>Uint8C</td>
<td>1</td>
<td>ToUint8Clamp</td>
<td>8-bit unsigned integer (clamped conversion)</td>
</tr>
<tr>
<td>Int16Array %Int16Array%</td>
<td>Int16</td>
<td>2</td>
<td>ToInt16</td>
<td>16-bit two’s complement signed integer</td>
</tr>
<tr>
<td>Uint16Array %Uint16Array%</td>
<td>Uint16</td>
<td>2</td>
<td>ToUint16</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>Int32Array %Int32Array%</td>
<td>Int32</td>
<td>4</td>
<td>ToInt32</td>
<td>32-bit two’s complement signed integer</td>
</tr>
<tr>
<td>Uint32Array %Uint32Array%</td>
<td>Uint32</td>
<td>4</td>
<td>ToUint32</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>BigInt64Array %BigInt64Array%</td>
<td>BigInt64</td>
<td>8</td>
<td>ToBigInt64</td>
<td>64-bit two’s complement signed integer</td>
</tr>
<tr>
<td>BigUint64Array %BigUint64Array%</td>
<td>BigUint64</td>
<td>8</td>
<td>ToBigUint64</td>
<td>64-bit unsigned integer</td>
</tr>
<tr>
<td>Float32Array %Float32Array%</td>
<td>Float32</td>
<td>4</td>
<td></td>
<td>32-bit IEEE floating point</td>
</tr>
<tr>
<td>Float64Array %Float64Array%</td>
<td>Float64</td>
<td>8</td>
<td></td>
<td>64-bit IEEE floating point</td>
</tr>
</tbody>
</table>

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.
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, Pk »)`.
   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]".
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 +0xFF.
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 +0xFF.
6. Let \( \text{offset} \) be \( O.[[\text{ByteOffset}]] \).
7. Return \( F(\text{offset}) \).

### 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 \( \text{len} \) be \( O.[[\text{ArrayLength}]] \).
4. Let relativeTarget be \? ToIntegerOrInfinity\( (target) \).
5. If relativeTarget is \(-\infty\), let \( \text{to} \) be 0.
6. Else if relativeTarget < 0, let \( \text{to} \) be \( \max(\text{len} + \text{relativeTarget}, 0) \).
7. Else, let \( \text{to} \) be \( \min(\text{relativeTarget}, \text{len}) \).
8. Let relativeStart be \? ToIntegerOrInfinity\( (\text{start}) \).
9. If relativeStart is \(-\infty\), let \( \text{from} \) be 0.
10. Else if relativeStart < 0, let \( \text{from} \) be \( \max(\text{len} + \text{relativeStart}, 0) \).
11. Else, let \( \text{from} \) be \( \min(\text{relativeStart}, \text{len}) \).
12. If \( \text{end} \) is undefined, let relativeEnd be \( \text{len} \); else let relativeEnd be \? ToIntegerOrInfinity\( (\text{end}) \).
13. If relativeEnd is \(-\infty\), let \( \text{final} \) be 0.
14. Else if relativeEnd < 0, let \( \text{final} \) be \( \max(\text{len} + \text{relativeEnd}, 0) \).
15. Else, let \( \text{final} \) be \( \min(\text{relativeEnd}, \text{len}) \).
16. Let \( \text{count} \) be \( \min(\text{final} - \text{from}, \text{len} - \text{to}) \).
17. If \( \text{count} \) > 0, then
   a. NOTE: The copying must be performed in a manner that preserves the bit-level encoding of the source data.
   b. Let \( \text{buffer} \) be \( O.[[\text{ViewedArrayBuffer}]] \).
   c. If IsDetachedBuffer\( (\text{buffer}) \) is true, throw a TypeError exception.
   d. Let typedArrayName be the String value of \( O.[[\text{TypedArrayName}]] \).
   e. Let elementSize be the Element Size value specified in Table 60 for typedArrayName.
   f. Let byteOffset be \( O.[[\text{ByteOffset}]] \).
   g. Let toByteIndex be \( \text{to} \times \text{elementSize} + \text{byteOffset} \).
   h. Let fromByteIndex be \( \text{from} \times \text{elementSize} + \text{byteOffset} \).
   i. Let countBytes be \( \text{count} \times \text{elementSize} \).
   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 \( \text{countBytes} \) > 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($k$).
   b. Let $kValue$ be ! Get($O$, $Pk$).
   c. Let testResult be ! ToBoolean(? Call(callbackfn, thisArg, « $kValue$, $Pk$, $O$ »)).
   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 \( \text{relativeStart} \) is \(-\infty\), let \( k \) be 0.
8. Else if \( \text{relativeStart} < 0 \), let \( k \) be \( \max(\text{len} + \text{relativeStart}, 0) \).
9. Else, let \( k \) be \( \min(\text{relativeStart}, \text{len}) \).
10. If \( \text{end} \) is \( \text{undefined} \), let \( \text{relativeEnd} \) be \( \text{len} \); else let \( \text{relativeEnd} \) be \( ? \, \text{ToIntegerOrInfinity}(\text{end}) \).
11. If \( \text{relativeEnd} \) is \(-\infty\), let \( \text{final} \) be 0.
12. Else if \( \text{relativeEnd} < 0 \), let \( \text{final} \) be \( \max(\text{len} + \text{relativeEnd}, 0) \).
13. Else, let \( \text{final} \) be \( \min(\text{relativeEnd}, \text{len}) \).
14. If \( \text{IsDetachedBuffer}(\text{O}[[\text{ViewedArrayBuffer}]]) \) is \text{true}, throw a \text{TypeError} exception.
15. Repeat, while \( k < \text{final} \),
   a. Let \( Pk \) be \( \text{ToString}(F(k)) \).
   b. Perform \( \text{Set}(\text{O}, Pk, \text{value}, \text{true}) \).
   c. Set \( k \) to \( k + 1 \).
16. Return \( \text{O} \).

23.2.3.9 \%TypedArray%\.prototype.filter ( \( \text{callbackfn} \), \( \text{thisArg} \) )

The interpretation and use of the arguments of \%TypedArray%\.prototype.filter\ are the same as for \text{Array.prototype.filter} as defined in 23.1.3.7.

When the \text{filter} method is called with one or two arguments, the following steps are taken:

1. Let \( \text{O} \) be the \text{this} value.
2. Perform \( ? \, \text{ValidateTypedArray}(\text{O}) \).
3. Let \( \text{len} \) be \( \text{O}[[\text{ArrayLength}]] \).
4. If \( \text{IsCallable}(\text{callbackfn}) \) is \text{false}, throw a \text{TypeError} exception.
5. Let \( \text{kept} \) be a new empty \text{List}.
6. Let \( k \) be 0.
7. Let \( \text{captured} \) be 0.
8. Repeat, while \( k < \text{len} \),
   a. Let \( Pk \) be \( \text{ToString}(F(k)) \).
   b. Let \( kValue \) be \( \text{Get}(\text{O}, Pk) \).
   c. Let \( \text{selected} \) be \( \text{ToBoolean}(? \, \text{Call}(\text{callbackfn}, \text{thisArg}, \langle kValue, F(k), \text{O} \rangle)) \).
   d. If \( \text{selected} \) is \text{true}, then
      i. Append \( kValue \) to the end of \( \text{kept} \).
      ii. Set \( \text{captured} \) to \( \text{captured} + 1 \).
   e. Set \( k \) to \( k + 1 \).
9. Let \( \text{A} \) be \( \text{TypedArraySpeciesCreate}(\text{O}, \langle F(\text{captured}) \rangle) \).
10. Let \( n \) be 0.
11. For each element \( e \) of \( \text{kept} \), do
    a. Perform \( \text{Set}(\text{A}, ! \text{ToString}(F(n)), e, \text{true}) \).
    b. Set \( n \) to \( n + 1 \).
12. Return \( \text{A} \).

This function is not generic. The \text{this} value must be an object with a [[TypedArrayName]] internal slot.

23.2.3.10 \%TypedArray%\.prototype.find ( \( \text{predicate} \), \( \text{thisArg} \) )

The interpretation and use of the arguments of \%TypedArray%\.prototype.find\ are the same as for \text{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 $k$.
   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 $k < \text{len}$,
   a. Let $P_k$ be $\text{ToString}($f$(k))$.
   b. Let $k\text{Value}$ be $\text{Get}(O, P_k)$.
   c. Perform ? $\text{Call}(\text{callbackfn}, this\text{Arg}, \langle k\text{Value}, f(k), O \rangle)$.
   d. Set $k$ to $k + 1$.
7. Return $\text{undefined}$.

This function is not generic. The $\text{this}$ 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 $\text{this}$ value.
2. Perform ? $\text{ValidateTypedArray}(O)$.
3. Let $\text{len}$ be $O$.[[ArrayLength]].
4. If $\text{len}$ is 0, return $\text{false}$.
5. Let $n$ be $\text{ToIntegerOrInfinity}(\text{fromIndex})$.
6. Assert: If $\text{fromIndex}$ is $\text{undefined}$, then $n$ is 0.
7. If $n$ is $+\infty$, return $\text{false}$.
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 $\text{len} + n$.
    b. If $k < 0$, set $k$ to 0.
11. Repeat, while $k < \text{len}$,
    a. Let $\text{elementK}$ be $\text{Get}(O, \text{ToString}(f(k)))$.
    b. If $\text{SameValueZero}(\text{searchElement}, \text{elementK})$ is $\text{true}$, return $\text{true}$.
    c. Set $k$ to $k + 1$.
12. Return $\text{false}$.

This function is not generic. The $\text{this}$ 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 $\text{this}$ value.
2. Perform ? $\text{ValidateTypedArray}(O)$.
3. Let $\text{len}$ be $O$.[[ArrayLength]].
4. If $\text{len}$ is 0, return $-1_{\text{f}}$.
5. Let $n$ be $\text{ToIntegerOrInfinity}(\text{fromIndex})$.
6. Assert: If $\text{fromIndex}$ is $\text{undefined}$, 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 $\text{len} + n$.
    b. If $k < 0$, set $k$ to 0.
11. Repeat, while $k < \text{len}$,
    a. Let $k\text{Present}$ be $\text{HasProperty}(O, \text{ToString}(\mathbb{F}(k)))$.
    b. If $k\text{Present}$ is true, then
       i. Let $elementK$ be $\text{Get}(O, \text{ToString}(\mathbb{F}(k)))$.
       ii. Let $same$ be the result of performing $\text{Strict Equality Comparison} searchElement === elementK$.
       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 $\text{this}$ value must be an object with a $[[\text{TypedArrayName}]]$ internal slot.

### 23.2.3.15 $\%\text{TypedArray}\%.prototype.join ( \text{separator} )$

The interpretation and use of the arguments of $\%\text{TypedArray}\%.prototype.join$ are the same as for $\text{Array.prototype.join}$ as defined in 23.1.3.15.

When the $\text{join}$ method is called with one argument $\text{separator}$, the following steps are taken:

1. Let $O$ be the $\text{this}$ value.
2. Perform $\text{ValidateTypedArray}(O)$.
3. Let $\text{len}$ be $O.[[\text{ArrayLength}]]$.
4. If $\text{separator}$ is $\text{undefined}$, let $sep$ be the single-element String ",".
5. Else, let $sep$ be $\text{ToString}(\text{separator})$.
6. Let $R$ be the empty String.
7. Let $k$ be 0.
8. Repeat, while $k < \text{len}$,
   a. If $k > 0$, set $R$ to the string-concatenation of $R$ and $sep$.
   b. Let $element$ be $\text{Get}(O, \text{ToString}(\mathbb{F}(k)))$.
   c. If $element$ is $\text{undefined}$, let $next$ be the empty String; otherwise, let $next$ be $\text{ToString}(element)$.
   d. Set $R$ to the string-concatenation of $R$ and $next$.
   e. Set $k$ to $k + 1$.

This function is not generic. The $\text{this}$ value must be an object with a $[[\text{TypedArrayName}]]$ internal slot.

### 23.2.3.16 $\%\text{TypedArray}\%.prototype.keys ()$

The following steps are taken:

1. Let $O$ be the $\text{this}$ 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 -1F.
5. If fromIndex is present, let n be ? ToIntegerOrInfinity(fromIndex); else let n be len - 1.
6. If n is -∞, return -1F.
7. If n ≥ 0, then
   a. Let k be min(n, len - 1).
8. Else,
   a. Let k be len + n.
9. 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 F(k).
   c. Set k to k - 1.
10. Return -1F.

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 +0F.
6. Let length be O.[[ArrayLength]].
7. Return F(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, « $ (len) »).
6. Let k be 0.
7. Repeat, while k < len,
   a. Let Pk be ! ToString($ (k)).
   b. Let kValue be ! Get(O, Pk).
   c. Let mappedValue be ? Call(callbackfn, thisArg, « kValue, $ (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($ (k)).
   b. Set accumulator to ! Get(O, Pk).
   c. Set k to k + 1.
10. Repeat, while k < len,
    a. Let Pk be ! ToString($ (k)).
    b. Let kValue be ! Get(O, Pk).
    c. Set accumulator to ? Call(callbackfn, undefined, « accumulator, kValue, $ (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.
This function is not generic. The this value must be an object with a [[TypedArrayName]] internal slot.

---

%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.

---

SetTypedArrayFromTypedArray ( target, targetOffset, source )

The abstract operation SetTypedArrayFromTypedArray takes arguments target (a TypedArray object), targetOffset (a non-negative integer or +∞), 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 typeName be the String value of target. [[TypedArrayName]].
8. Let targetType be the Element Type value in Table 60 for typeName.
9. Let targetElementSize be the Element Size value specified in Table 60 for typeName.
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 +∞, throw a RangeError exception.
17. If srcLength + targetOffset > targetLength, throw a RangeError exception.
18. If target. [[ContentType]] ≠ 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 is it 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 +∞), 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 targetType be the Element Type value 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 +∞, 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 $\text{ToString}(k)$.
   b. Let $value$ be $\text{Get}(src, Pk)$.
   c. If $target$.[[ContentType]] is BigInt, set $value$ to $\text{ToBigInt}(value)$.
   d. Otherwise, set $value$ to $\text{ToNumber}(value)$.
   e. If $\text{IsDetachedBuffer}(targetBuffer)$ is true, throw a TypeError exception.
   f. Perform $\text{SetValueInBuffer}(targetBuffer, targetByteIndex, targetType, value, true, Unordered)$.
   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 $\text{ValidateTypedArray}(O)$.
3. Let $len$ be $O$.[[ArrayLength]].
4. Let $relativeStart$ be $\text{ToIntegerOrInfinity}(start)$.
5. If $relativeStart$ is $-\infty$, let $k$ be 0.
6. Else if $relativeStart < 0$, let $k$ be max($len + relativeStart$, 0).
7. Else, let $k$ be min($relativeStart$, $len$).
8. If $end$ is undefined, let $relativeEnd$ be $len$; else let $relativeEnd$ be $\text{ToIntegerOrInfinity}(end)$.
9. If $relativeEnd$ is $-\infty$, let $final$ be 0.
10. Else if $relativeEnd < 0$, let $final$ be max($len + relativeEnd$, 0).
11. Else, let $final$ be min($relativeEnd$, $len$).
12. Let $count$ be max($final - k$, 0).
13. Let $A$ be $\text{TypedArraySpeciesCreate}(O, « F(count) »)$.
14. If $count > 0$, then
   a. If $\text{IsDetachedBuffer}(O$.[[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 $\text{ToString}(F(k))$.
          2. Let $kValue$ be $\text{Get}(O, Pk)$.
          3. Perform $\text{Set}(A, ! \text{ToString}(F(n)), kValue, true)$.
          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 $O_{\text{[ByteOffset]}}$.
v. Let targetByteIndex be $A_{\text{[ByteOffset]}}$.
vii. Let srcByteIndex be $(k \times \text{elementSize}) + \text{srcByteOffset}$.
viii. Let limit be targetByteIndex + count $\times$ elementSize.
ix. Repeat, while targetByteIndex < limit,
   1. Let value be GetValueFromBuffer(srcBuffer, srcByteIndex, Uint8, true, Unordered).
   2. Perform SetValueInBuffer(targetBuffer, targetByteIndex, Uint8, value, true, Unordered).
   3. Set srcByteIndex to srcByteIndex + 1.
   4. Set targetByteIndex to targetByteIndex + 1.

15. Return $A$.

This function is not generic. The this value must be an object with a $[[\text{TypedArrayName}]]$ 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 $k < len$,
   a. Let $Pk$ be ! ToString($k$).
   b. Let kValue be ! Get(O, $Pk$).
   c. Let testResult be ! ToBoolean(? Call(callbackfn, thisArg, « kValue, $Pk$, O »)).
   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 $[[\text{TypedArrayName}]]$ 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 $[[\text{TypedArrayName}]]$ 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 \( \text{len} \) be \( \text{obj}[[\text{ArrayLength}]] \).

The following version of \text{SortCompare} is used by \%\text{TypedArray}\%.\text{prototype}.\text{sort}. It performs a numeric comparison rather than the string comparison used in 23.1.3.27.

The abstract operation \text{TypedArraySortCompare} takes arguments \( x \) and \( y \). It also has access to the \text{comparefn} and \text{buffer} values of the current invocation of the \text{sort} method. It performs the following steps when called:

1. \textbf{Assert:} Both \text{Type}(x) and \text{Type}(y) are Number or both are BigInt.
2. If \text{comparefn} is not \text{undefined}, then
   a. Let \( v \) be ? \text{ToNumber}(\text{Call}(\text{comparefn}, \text{undefined}, \langle x, y \rangle))
   b. If \text{IsDetachedBuffer}(\text{buffer}) is \text{true}, throw a \text{TypeError} exception.
   c. If \( v \) is \text{NaN}, return \( +0 \).
   d. Return \( v \).
3. If \( x \) and \( y \) are both \text{NaN}, return \( +0 \).
4. If \( x \) is \text{NaN}, return \( 1 \).
5. If \( y \) is \text{NaN}, return \( -1 \).
6. If \( x < y \), return \( -1 \).
7. If \( x > y \), return \( 1 \).
8. If \( x \) is \( -0 \) and \( y \) is \( +0 \), return \( -1 \).
9. If \( x \) is \( +0 \) and \( y \) is \( -0 \), return \( 1 \).
10. Return \( +0 \).

\textbf{NOTE} Because \text{NaN} always compares greater than any other value, \text{NaN} property values always sort to the end of the result when \text{comparefn} is not provided.

\textit{23.2.3.27} \%\text{TypedArray}%.\text{prototype}.\text{subarray} ( \text{begin}, \text{end} )

Returns a new \text{TypedArray} object whose element type is the same as this \text{TypedArray} and whose ArrayBuffer is the same as the ArrayBuffer of this \text{TypedArray}, referencing the elements at \text{begin}, inclusive, up to \text{end}, exclusive. If either \text{begin} or \text{end} is negative, it refers to an index from the end of the array, as opposed to from the beginning.

1. \textbf{Let} \( O \) be the this value.
2. Perform ? \text{RequireInternalSlot}(O, [[\text{TypedArrayName}]]).
3. \textbf{Assert:} \( O \) has a [[\text{ViewedArrayBuffer}]] internal slot.
4. Let \text{buffer} be \( O[[\text{ViewedArrayBuffer}]] \).
5. Let \text{srcLength} be \( O[[\text{ArrayLength}]] \).
6. Let \text{relativeBegin} be ? \text{ToIntegerOrInfinity}(\text{begin}).
7. If \text{relativeBegin} is \(-\infty\), let \text{beginIndex} be \( 0 \).
8. Else if \text{relativeBegin} < \( 0 \), let \text{beginIndex} be \( \max(\text{srcLength} + \text{relativeBegin}, 0) \).
9. Else, let \text{beginIndex} be \( \min(\text{relativeBegin}, \text{srcLength}) \).
10. If \text{end} is \text{undefined}, let \text{relativeEnd} be \text{srcLength}; else let \text{relativeEnd} be ? \text{ToIntegerOrInfinity}(\text{end}).
11. If \text{relativeEnd} is \(-\infty\), let \text{endIndex} be \( 0 \).
12. Else if \text{relativeEnd} < \( 0 \), let \text{endIndex} be \( \max(\text{srcLength} + \text{relativeEnd}, 0) \).
13. Else, let \text{endIndex} be \( \min(\text{relativeEnd}, \text{srcLength}) \).
14. Let \text{newLength} be \( \max(\text{endIndex} - \text{beginIndex}, 0) \).
15. Let \text{constructorName} be the String value of \( O[[\text{TypedArrayName}]] \).
16. Let \( \text{elementSize} \) be the Element Size value specified in Table 60 for \( \text{constructorName} \).

17. Let \( \text{srcByteOffset} \) be \( O.[[\text{ByteOffset}]] \).

18. Let \( \text{beginByteOffset} \) be \( \text{srcByteOffset} + \text{beginIndex} \times \text{elementSize} \).

19. Let \( \text{argumentsList} \) be « \( \text{buffer, F(beginByteOffset), F(newLength)} \) ».

20. Return \( \text{TypedArraySpeciesCreate}(O, \text{argumentsList}) \).

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\( (O, \text{value}) \).

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.\([\text{[TypedArrayName]}]\).
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.\([\text{[ContentType]}] \neq \) exemplar.\([\text{[ContentType]}]\), 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).
3. If argumentList is a List of a single Number, then
   a. If newTypedArray.\([\text{[ArrayLength]}] < \) \(\mathbb{R}(\) 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 \(3\). 

### 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 an `[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 an `[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 $\text{AllocateTypedArray}(\text{constructorName}, \text{NewTarget}, \text{proto}, \text{elementLength})$.

### 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 $\text{GetPrototypeOf}(\text{newTarget}, \text{defaultProto})$.
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 $\text{AllocateTypedArrayBuffer}(\text{obj}, \text{length})$.

### 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` $\times$ `elementLength`.
14. If `IsSharedArrayBuffer(srcData)` is `false`, then
   a. Let `bufferConstructor` be $\text{SpeciesConstructor}(\text{srcData}, \%\text{ArrayBuffer}%)$.
15. Else,
   a. Let `bufferConstructor` be `\%\text{ArrayBuffer}%`.
16. If `elementType` is the same as `srcType`, then
   a. Let `data` be $\text{CloneArrayBuffer}(\text{srcData}, \text{srcByteOffset}, \text{byteLength}, \text{bufferConstructor})$.
17. Else,
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 \([\text{TypedArrayName}]\) internal slot.
2. Assert: \( buffer \) is an Object that has an \([\text{ArrayBufferData}]\) internal slot.
3. Let \( constructorName \) be the String value of \( O.[[\text{TypedArrayName}]] \).
4. Let \( elementSize \) be the Element Size value specified in Table 60 for \( constructorName \).
5. Let \( offset \) be \( ? \, \text{ToIndex}(byteOffset) \).
6. If \( offset \) modulo \( elementSize \) \( \neq \) 0, throw a \( \text{RangeError} \) exception.
7. If \( length \) is not \( \text{undefined} \), then
   a. Let \( newLength \) be \( ? \, \text{ToIndex}(length) \).
   b. If \( IsDetachedBuffer(buffer) \) is \( \text{true} \), throw a \( \text{TypeError} \) exception.
9. Let \( bufferByteLength \) be \( buffer.[[\text{ArrayBufferByteLength}]] \).
10. If \( length \) is \( \text{undefined} \), then
    a. If \( bufferByteLength \) modulo \( elementSize \) \( \neq \) 0, throw a \( \text{RangeError} \) exception.
    b. Let \( newByteLength \) be \( bufferByteLength \) \( - \) \( offset \).
    c. If \( newByteLength \) \( < \) 0, throw a \( \text{RangeError} \) exception.
11. Else,
    a. Let \( newByteLength \) be \( newLength \times \) \( elementSize \).
    b. If \( offset + newByteLength > bufferByteLength \), throw a \( \text{RangeError} \) exception.
12. Set \( O.[[\text{ViewedArrayBuffer}]] \) to \( buffer \).
13. Set \( O.[[\text{ByteLength}]] \) to \( newByteLength \).
14. Set \( O.[[\text{ByteOffset}]] \) to \( offset \).
15. Set \( O.[[\text{ArrayLength}]] \) to \( newByteLength \) \( / \) \( elementSize \).

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 $values$.
4. Let $k$ be 0.
5. Repeat, while $k < len$,
   a. Let $Pk$ be ! ToString($k$).
   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($arrayLike$).
4. Let $k$ be 0.
5. Repeat, while $k < len$,
   a. Let $Pk$ be ! ToString($k$).
   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 $elementSize$ × $length$.
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").

**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(addar, 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).

The existing [[MapData]] List is preserved because there may be existing Map Iterator objects that are suspended midway through iterating over that List.
24.1.3.5 Map.prototype.forEach ( \( \text{callbackfn} \ [ , \ \text{thisArg} \ ] \) )

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 ( \( \text{key} \) )

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 ( \( \text{key} \) )

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]]`.
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\( \bigcirc \), set \( key \) to +0\( \bigcirc \).
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 \( \bigcirc \)\( (count) \).

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 \texttt{RequireInternalSlot(S, \texttt{[[SetData]]}).}
3. Let \texttt{entries} be the \texttt{List} that is \texttt{S.\texttt{[[SetData]]].}
4. For each element \texttt{e} of \texttt{entries}, do
   a. If \texttt{e} is not \texttt{empty} and \texttt{SameValueZero(e, value)} is \texttt{true}, then
      i. Return \texttt{S}.
5. If \texttt{value} is \texttt{-0}, set \texttt{value} to \texttt{+0}.
6. Append \texttt{value} as the last element of \texttt{entries}.
7. Return \texttt{S}.

\textbf{24.2.3.2 \texttt{Set.prototype.clear()} }

The following steps are taken:

1. Let \texttt{S} be the \texttt{this} value.
2. Perform \texttt{RequireInternalSlot(S, \texttt{[[SetData]]}).}
3. Let \texttt{entries} be the \texttt{List} that is \texttt{S.\texttt{[[SetData]]].}
4. For each element \texttt{e} of \texttt{entries}, do
   a. Replace the element of \texttt{entries} whose value is \texttt{e} with an element whose value is \texttt{empty}.
5. Return \texttt{undefined}.

\textbf{NOTE}  
The existing \texttt{[[SetData]]} \texttt{List} is preserved because there may be existing Set Iterator objects that are suspended midway through iterating over that \texttt{List}.

\textbf{24.2.3.3 \texttt{Set.prototype.constructor} }

The initial value of \texttt{\texttt{Set.prototype.constructor}} is \texttt{%Set\%}.

\textbf{24.2.3.4 \texttt{Set.prototype.delete(value)} }

The following steps are taken:

1. Let \texttt{S} be the \texttt{this} value.
2. Perform \texttt{RequireInternalSlot(S, \texttt{[[SetData]]}).}
3. Let \texttt{entries} be the \texttt{List} that is \texttt{S.\texttt{[[SetData]]].}
4. For each element \texttt{e} of \texttt{entries}, do
   a. If \texttt{e} is not \texttt{empty} and \texttt{SameValueZero(e, value)} is \texttt{true}, then
      i. Replace the element of \texttt{entries} whose value is \texttt{e} with an element whose value is \texttt{empty}.
      ii. Return \texttt{true}.
5. Return \texttt{false}.

\textbf{NOTE}  
The value \texttt{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.

\textbf{24.2.3.5 \texttt{Set.prototype.entries()}}

The following steps are taken:
1. Let \( S \) be the this value.
2. Return \( \text{CreateSetIterator}(S, \text{key+value}) \).

**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 \( \text{RequireInternalSlot}(S, [[SetData]]) \).
3. If IsCallable(callbackfn) is false, throw a TypeError exception.
4. Let \( \text{entries} \) be the List that is \( S.\text{[[SetData]]} \).
5. For each element \( e \) of \( \text{entries} \), do
   a. If \( e \) is not empty, then
      i. Perform \( \text{Call(callbackfn, thisArg, « e, e, S »)} \).
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 $\Phi(count)$.

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:


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(\( M, [[\text{WeakMapData}]] \)).
3. Let \( \text{entries} \) be the List that is \( M.\![\![[\text{WeakMapData}]]\!] \).
4. If Type(\( \key \)) is not Object, return false.
5. For each Record \( \{ [[\text{Key}}], [[\text{Value}}]] \} \ p \) of \( \text{entries} \), do
   a. If \( p.\![\![[\text{Key}}]\!] \) is not empty and SameValue(\( p.\![\![[\text{Key}}]\!], \key \)) is true, then
      i. Set \( p.\![\![[\text{Key}}]\!] \) to empty.
      ii. Set \( p.\![\![[\text{Value}}]\!] \) 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(\( M, [[\text{WeakMapData}]] \)).
3. Let \( \text{entries} \) be the List that is \( M.\![\![[\text{WeakMapData}]]\!] \).
4. If Type(\( \key \)) is not Object, return undefined.
5. For each Record \( \{ [[\text{Key}}], [[\text{Value}}]] \} \ p \) of \( \text{entries} \), do
   a. If \( p.\![\![[\text{Key}}]\!] \) is not empty and SameValue(\( p.\![\![[\text{Key}}]\!], \key \)) is true, return \( p.\![\![[\text{Value}}]\!] \).
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(\( M, [[\text{WeakMapData}]] \)).
3. Let \( \text{entries} \) be the List that is \( M.\![\![[\text{WeakMapData}]]\!] \).
4. If Type(\( \key \)) is not Object, return false.
5. For each Record \( \{ [[\text{Key}}], [[\text{Value}}]] \} \ p \) of \( \text{entries} \), do
   a. If \( p.\![\![[\text{Key}}]\!] \) is not empty and SameValue(\( p.\![\![[\text{Key}}]\!], \key \)) 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(\( M, [[\text{WeakMapData}]] \)).
3. Let \( \text{entries} \) be the List that is \( M.\![\![[\text{WeakMapData}]]\!] \).
4. If `Type(key)` is not `Object`, throw a `TypeError` exception.

5. For each `Record` `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, 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]]: \textbf{false}, 
[[Enumerable]]: \textbf{false}, 
[[Configurable]]: \textbf{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 \textit{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.

\textbf{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 (\textit{constructor}, \textit{byteLength})

The abstract operation \texttt{AllocateArrayBuffer} takes arguments \textit{constructor} and \textit{byteLength} (a non-negative integer). It is used to create an ArrayBuffer object. It performs the following steps when called:

1. Let \texttt{obj} be ? \texttt{OrdinaryCreateFromConstructor} (\textit{constructor}, "%ArrayBuffer.prototype%", « [[ArrayBufferData]], [[ArrayBufferByteLength]], [[ArrayBufferDetachKey]] »).
2. Let \texttt{block} be ? \texttt{CreateByteDataBlock} (\textit{byteLength}).
3. Set \texttt{obj}:[[ArrayBufferData]] to \texttt{block}.
4. Set \texttt{obj}:[[ArrayBufferByteLength]] to \textit{byteLength}.
5. Return \texttt{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 `BigInt64` or `BigUint64`, 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
7. If ! IsBigIntElementType\(\text{(type)}\) is true, return the BigInt value that corresponds to \(\text{intValue}\).
8. Otherwise, return the Number value that corresponds to \(\text{intValue}\).

25.1.2.10  GetValueFromBuffer ( \text{arrayBuffer}, \text{byteIndex}, \text{type}, \text{isTypedArray}, \text{order} [ , \text{isLittleEndian} ] )

The abstract operation GetValueFromBuffer takes arguments \text{arrayBuffer} (an ArrayBuffer or SharedArrayBuffer), \text{byteIndex} (a non-negative integer), \text{type} (a TypedArray element type), \text{isTypedArray} (a Boolean), and \text{order} (either SeqCst or Unordered) and optional argument \text{isLittleEndian} (a Boolean). It performs the following steps when called:

1. Assert: IsDetachedBuffer\(\text{(arrayBuffer)}\) is false.
2. Assert: There are sufficient bytes in \text{arrayBuffer} starting at \text{byteIndex} to represent a value of \text{type}.
3. Let \text{block} be \text{arrayBuffer}[[ArrayBufferData]].
4. Let \text{elementSize} be the Element Size value specified in Table 60 for Element Type \text{type}.
5. If IsSharedArrayBuffer\(\text{(arrayBuffer)}\) is true, then
   a. Let \text{execution} be the [[CandidateExecution]] field of the surrounding agent's Agent Record.
   b. Let \text{eventList} be the [[EventList]] field of the element in \text{execution}[[EventsRecords]] whose [[AgentSignifier]] is AgentSignifier().
   c. If \text{isTypedArray} is true and IsNoTearConfiguration\(\text{(type, order)}\) is true, let \text{noTear} be true; otherwise let \text{noTear} be false.
   d. Let \text{rawValue} be a List of length \text{elementSize} whose elements are nondeterministically chosen byte values.
   e. NOTE: In implementations, \text{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 \text{readEvent} be ReadSharedMemory \{ [[Order]]: \text{order}, [[NoTear]]: \text{noTear}, [[Block]]: \text{block}, [[ByteIndex]]: \text{byteIndex}, [[ElementSize]]: \text{elementSize} \}.
   g. Append \text{readEvent} to \text{eventList}.
   h. Append Chosen Value Record \{ [[Event]]: \text{readEvent}, [[ChosenValue]]: \text{rawValue} \} to \text{execution}.
5. Else, let \text{rawValue} be a List whose elements are bytes from \text{block} at indices \text{byteIndex} (inclusive) through \text{byteIndex} + \text{elementSize} (exclusive).
6. Assert: The number of elements in \text{rawValue} is \text{elementSize}.
7. If \text{isLittleEndian} is not present, set \text{isLittleEndian} to the value of the [[LittleEndian]] field of the surrounding agent's Agent Record.
9. Return RawBytesToNumeric\(\text{(type, rawValue, isLittleEndian)}\).

25.1.2.11  NumericToRawBytes ( \text{type}, \text{value}, \text{isLittleEndian} )

The abstract operation NumericToRawBytes takes arguments \text{type} (a TypedArray element type), \text{value} (a BigInt or a Number), and \text{isLittleEndian} (a Boolean). It performs the following steps when called:

1. If \text{type} is Float32, then
   a. Let \text{rawBytes} be a List whose elements are the 4 bytes that are the result of converting \text{value} to IEEE 754-2019 binary32 format using roundTiesToEven mode. If \text{isLittleEndian} is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If \text{value} is NaN, \text{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 `ℝ(convOp(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 \( \text{arrayBuffer} \) (an ArrayBuffer object or a SharedArrayBuffer object), \( \text{byteIndex} \) (a non-negative integer), \( \text{type} \) (a TypedArray element type), \( \text{value} \) (a Number or a BigInt), and \( \text{op} \) (a read-modify-write modification function) and optional argument \( \text{isLittleEndian} \) (a Boolean). It performs the following steps when called:

1. Assert: \( \text{IsDetachedBuffer} (\text{arrayBuffer}) \) is false.
2. Assert: There are sufficient bytes in \( \text{arrayBuffer} \) starting at \( \text{byteIndex} \) to represent a value of \( \text{type} \).
3. Assert: \( \text{Type} (\text{value}) \) is BigInt if \( \text{IsBigIntElementType} (\text{type}) \) is true; otherwise, \( \text{Type} (\text{value}) \) is Number.
4. Let \( \text{block} \) be \( \text{arrayBuffer}.[[\text{ArrayBufferData}]] \).
5. Let \( \text{elementSize} \) be the Element Size value specified in Table 60 for Element Type \( \text{type} \).
6. If \( \text{isLittleEndian} \) is not present, set \( \text{isLittleEndian} \) to the value of the [[LittleEndian]] field of the surrounding agent’s Agent Record.
7. Let \( \text{rawBytes} \) be \( \text{NumericToRawBytes} (\text{type}, \text{value}, \text{isLittleEndian}) \).
8. If \( \text{IsSharedArrayBuffer} (\text{arrayBuffer}) \) is true, then
   a. Let \( \text{execution} \) be the [[CandidateExecution]] field of the surrounding agent’s Agent Record.
   b. Let \( \text{eventList} \) be the [[EventList]] field of the element in \( \text{execution}.[[\text{EventsRecords}]] \) whose [[AgentSignifier]] is \( \text{AgentSignifier}() \).
   c. Let \( \text{rawBytesRead} \) be a List of length \( \text{elementSize} \) whose elements are nondeterministically chosen byte values.
   d. NOTE: In implementations, \( \text{rawBytesRead} \) 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 \( \text{rmwEvent} \) be \( \text{ReadModifyWriteSharedMemory} \) \{ [[Order]]: \text{SeqCst}, [[NoTear]]: true, [[Block]]: \text{block}, [[ByteIndex]]: \text{byteIndex}, [[ElementSize]]: \text{elementSize}, [[Payload]]: \text{rawBytes}, [[ModifyOp]]: \text{op} \}.
   f. Append \( \text{rmwEvent} \) to \( \text{eventList} \).
   g. Append Chosen Value Record \{ [[Event]]: \text{rmwEvent}, [[ChosenValue]]: \text{rawBytesRead} \} to \( \text{execution}.[[\text{ChosenValues}]] \).
9. Else,
   a. Let \( \text{rawBytesRead} \) be a List of length \( \text{elementSize} \) whose elements are the sequence of \( \text{elementSize} \) bytes starting with \( \text{block}[\text{byteIndex}] \).
   b. Let \( \text{rawBytesModified} \) be \( \text{op}(\text{rawBytesRead}, \text{rawBytes}) \).
   c. Store the individual bytes of \( \text{rawBytesModified} \) into \( \text{block} \), starting at \( \text{block}[\text{byteIndex}] \).
10. Return \( \text{RawBytesToNumeric} (\text{type}, \text{rawBytesRead}, \text{isLittleEndian}) \).

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).

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)`.

### 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 `/u1D53D(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 `~0`, 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 `~0`, 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`, `{ `/u1D53D(newLen)` }).
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]]`.

719
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.
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`.

### 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 \( \text{F}(\text{size}) \).

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 $[[\text{ViewedArrayBuffer}]]$ internal slot.
4. Let $buffer$ be $O.\\![[\text{ViewedArrayBuffer}]]$.
5. If $\text{IsDetachedBuffer}(buffer)$ is true, throw a $\text{TypeError}$ exception.
6. Let $offset$ be $O.\\![[\text{ByteOffset}]]$.
7. Return $F(offset)$.

### 25.3.4.4 DataView.prototype.constructor

The initial value of $\text{DataView.prototype.constructor}$ is $%\text{DataView}%$.

### 25.3.4.5 DataView.prototype.getBigInt64 ($byteOffset$, $littleEndian$)

When the $\text{getBigInt64}$ method is called with argument $byteOffset$ and optional argument $littleEndian$, the following steps are taken:

1. Let $v$ be the this value.
2. Return $\text{GetViewValue}(v, byteOffset, littleEndian, \text{BigInt64})$.

### 25.3.4.6 DataView.prototype.getBigUint64 ($byteOffset$, $littleEndian$)

When the $\text{getBigUint64}$ method is called with argument $byteOffset$ and optional argument $littleEndian$, the following steps are taken:

1. Let $v$ be the this value.
2. Return $\text{GetViewValue}(v, byteOffset, littleEndian, \text{BigUint64})$.

### 25.3.4.7 DataView.prototype.getFloat32 ($byteOffset$, $littleEndian$)

When the $\text{getFloat32}$ 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 $\text{false}$.
3. Return $\text{GetViewValue}(v, byteOffset, littleEndian, \text{Float32})$.

### 25.3.4.8 DataView.prototype.getFloat64 ($byteOffset$, $littleEndian$)

When the $\text{getFloat64}$ 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 $\text{false}$.
3. Return $\text{GetViewValue}(v, byteOffset, littleEndian, \text{Float64})$.

### 25.3.4.9 DataView.prototype.getInt8 ($byteOffset$)

When the $\text{getInt8}$ 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.

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.

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.

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.

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 \texttt{setBigUint64} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian}, the following steps are taken:

1. Let \( v \) be the this value.
2. Return ? SetViewValue(\( v \), byteOffset, littleEndian, BigUint64, value).

### 25.3.4.17 DataView.prototype.setFloat32 (byteOffset, value [, littleEndian ])

When the \texttt{setFloat32} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian}, the following steps are taken:

1. Let \( v \) be the this value.
2. If \texttt{littleEndian} is not present, set \texttt{littleEndian} to \texttt{false}.

### 25.3.4.18 DataView.prototype.setFloat64 (byteOffset, value [, littleEndian ])

When the \texttt{setFloat64} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian}, the following steps are taken:

1. Let \( v \) be the this value.
2. If \texttt{littleEndian} is not present, set \texttt{littleEndian} to \texttt{false}.
3. Return ? SetViewValue(\( v \), byteOffset, littleEndian, Float64, value).

### 25.3.4.19 DataView.prototype.setInt8 (byteOffset, value )

When the \texttt{setInt8} method is called with arguments \texttt{byteOffset} and \texttt{value}, the following steps are taken:

1. Let \( v \) be the this value.
2. Return ? SetViewValue(\( v \), byteOffset, true, Int8, value).

### 25.3.4.20 DataView.prototype.setInt16 (byteOffset, value [, littleEndian ])

When the \texttt{setInt16} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian}, the following steps are taken:

1. Let \( v \) be the this value.
2. If \texttt{littleEndian} is not present, set \texttt{littleEndian} to \texttt{false}.

### 25.3.4.21 DataView.prototype.setInt32 (byteOffset, value [, littleEndian ])

When the \texttt{setInt32} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian}, the following steps are taken:
1. Let \( v \) be the this value.
2. If littleEndian is not present, set littleEndian to false.
3. Return \( \text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{Uint32}, \text{value}) \).

### 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 \( \text{SetViewValue}(v, \text{byteOffset}, \text{true}, \text{Uint8}, \text{value}) \).

### 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 \( \text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{Uint16}, \text{value}) \).

### 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 \( \text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{Uint32}, \text{value}) \).

### 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 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 ≥ 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 × elementSize) + offset.
A *WaiterList* is a semantic object that contains an ordered list of those agents that are waiting on a location \((\text{block}, i)\) in shared memory; \textit{block} is a Shared Data Block and \(i\) a byte offset into the memory of \textit{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 \((\text{block}, i)\). WaiterLists are agent-independent: a lookup in the store of WaiterLists by \((\text{block}, i)\) 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 \textit{block} (a Shared Data Block) and \(i\) (a non-negative integer). It performs the following steps when called:

1. **Assert:** \textit{block} is a Shared Data Block.
2. **Assert:** \(i\) and \(i + 3\) are valid byte offsets within the memory of \textit{block}.
3. **Assert:** \(i\) is divisible by 4.
4. Return the WaiterList that is referenced by the pair \((\text{block}, i)\).

The abstract operation EnterCriticalSection takes argument \(\text{WL}\) (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 \(\text{WL}\), then enter the critical section for \(\text{WL}\) (without allowing any other agent to enter).
3. If \(\text{WL}\) has a Synchronize event, then
   a. NOTE: A \(\text{WL}\) whose critical section has been entered at least once has a Synchronize event set by LeaveCriticalSection.
   b. Let \textit{execution} be the \[[\text{CandidateExecution}]] field of the surrounding agent's Agent Record.
   c. Let \textit{eventsRecord} be the Agent Events Record in \textit{execution}.[[EventsRecords]] whose [[AgentSignifier]] is AgentSignifier().
   d. Let \textit{entererEventList} be \textit{eventsRecord}.[[EventList]].
   e. Let \textit{enterEvent} be a new Synchronize event.
   f. Append \textit{enterEvent} to \textit{entererEventList}.
   g. Let \textit{leaveEvent} be the Synchronize event in \(\text{WL}\).
   h. Append \((\text{leaveEvent}, \text{enterEvent})\) to \textit{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 +∞). 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 \( \text{timeout} \) (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 \( \text{timeout} \) 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 ( \( \text{typedArray}, \text{index}, \text{value}, \text{op} \) )**

The abstract operation AtomicReadModifyWrite takes arguments \( \text{typedArray} \), \( \text{index} \), \( \text{value} \), and \( \text{op} \) (a read-modify-write modification function). \( \text{op} \) takes two \( \text{List} \) of byte values arguments and returns a \( \text{List} \) 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 \( \text{buffer} \) be ? ValidateIntegerTypedArray(\( \text{typedArray} \)).
2. Let \( \text{indexedPosition} \) be ? ValidateAtomicAccess(\( \text{typedArray}, \text{index} \)).
3. Let \( \text{arrayTypeName} \) be \( \text{typedArray}.[[\text{TypedArrayName}]] \).
4. If \( \text{typedArray}.[[\text{ContentType}]] \) is BigInt, let \( v \) be ? ToBigInt(\( \text{value} \)).
5. Otherwise, let \( v \) be ? ToIntegerOrInfinity(\( \text{value} \)).
6. If IsDetachedBuffer(\( \text{buffer} \)) 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 \( \text{elementType} \) be the Element Type value in Table 60 for \( \text{arrayTypeName} \).
9. Return GetModifySetValueInBuffer(\( \text{buffer}, \text{indexedPosition}, \text{elementType}, v, \text{op} \)).

25.4.1.12 **ByteListBitwiseOp ( \( \text{op}, \text{xBytes}, \text{yBytes} \) )**

The abstract operation ByteListBitwiseOp takes arguments \( \text{op} \) (a sequence of Unicode code points), \( \text{xBytes} \) (a \( \text{List} \) of
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: \( \text{op} \) is \& \&, ^ ^, or | |.
2. Assert: \( \text{xBytes} \) and \( \text{yBytes} \) have the same number of elements.
3. Let \( \text{result} \) be a new empty List.
4. Let \( i \) be 0.
5. For each element \( \text{xByte} \) of \( \text{xBytes} \), do
   a. Let \( \text{yByte} \) be \( \text{yBytes}[i] \).
   b. If \( \text{op} \) is \& \&, let \( \text{resultByte} \) be the result of applying the bitwise AND operation to \( \text{xByte} \) and \( \text{yByte} \).
   c. Else if \( \text{op} \) is ^ ^, let \( \text{resultByte} \) be the result of applying the bitwise exclusive OR (XOR) operation to \( \text{xByte} \) and \( \text{yByte} \).
   d. Else, \( \text{op} \) is | |. Let \( \text{resultByte} \) be the result of applying the bitwise inclusive OR operation to \( \text{xByte} \) and \( \text{yByte} \).
   e. Set \( i \) to \( i + 1 \).
   f. Append \( \text{resultByte} \) to the end of \( \text{result} \).
6. Return \( \text{result} \).

25.4.1.13 ByteListEqual ( \text{xBytes, yBytes} )

The abstract operation ByteListEqual takes arguments \( \text{xBytes} \) (a List of byte values) and \( \text{yBytes} \) (a List of byte values). It performs the following steps when called:

1. If \( \text{xBytes} \) and \( \text{yBytes} \) do not have the same number of elements, return false.
2. Let \( i \) be 0.
3. For each element \( \text{xByte} \) of \( \text{xBytes} \), do
   a. Let \( \text{yByte} \) be \( \text{yBytes}[i] \).
   b. If \( \text{xByte} \neq \text{yByte} \), return false.
   c. Set \( i \) to \( i + 1 \).
4. Return true.

25.4.2 Atomics.add ( \text{typedArray, index, value} )

The following steps are taken:

1. Let \( \text{type} \) be the Element Type value in Table 60 for \text{typedArray}[[TypedArrayName]].
2. Let \( \text{isLittleEndian} \) be the value of the [[LittleEndian]] field of the surrounding agent's Agent Record.
3. Let \( \text{add} \) be a new read-modify-write modification function with parameters \( \text{xBytes, yBytes} \) that captures \( \text{type} \) and \( \text{isLittleEndian} \) and performs the following steps atomically when called:
   a. Let \( x \) be \text{RawBytesToNumeric(type, xBytes, isLittleEndian)}.
   b. Let \( y \) be \text{RawBytesToNumeric(type, yBytes, isLittleEndian)}.
   c. Let \( T \) be \text{Type(x)}.
   d. Let \( \text{sum} \) be \( T::\text{add}(x, y) \).
   e. Let \( \text{sumBytes} \) be \text{NumericToRawBytes(type, sum, isLittleEndian)}.
   f. Assert: \( \text{sumBytes, xBytes, and yBytes} \) have the same number of elements.
   g. Return \( \text{sumBytes} \).
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]].
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 \(\#\) `ToIntegerOrInfinity(expectedValue)`.
   b. Let `replacement` be \(\#\) `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`.  

731
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.
**Atoms.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 **Atoms.isLockFree**(\( n \)) will return **true**. High-performance algorithms will use **Atoms.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.

**Atoms.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 **Atoms.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**.

### 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**(1, **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 "**BigInt64Array**" or "**BigUint64Array**", let \( v \) be ? **ToBigInt**(**value**).
5. Otherwise, let \( v \) be \( \frac{\text{ToIntegerOrInfinity}(\text{**value**})}{1} \).
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 `SetValuInBuffer(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 `+∞`, let `t` be `+∞`; else if `q` is `-∞`, let `t` be `0`; else let `t` be `max(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` ≠ `w`, then
   a. Perform `LeaveCriticalSection(WL)`.
   b. Return the String "not-equal".
17. Let \( W \) be \text{AgentSignifier}().
19. Let \( \text{notified} \) be \text{SuspendAgent}(WL, W, t).
20. If \( \text{notified} \) is \text{true}, then
   a. \text{Assert}: \( W \) is not on the list of waiters in \( WL \).
21. Else,
22. Perform \text{LeaveCriticalSection}(WL).
23. If \( \text{notified} \) is \text{true}, return the String "ok".
24. Return the String "timed-out".

25.4.12 \text{Atomics.notify} ( \text{typedArray, index, count} )

\text{Atomics.notify} notifies some agents that are sleeping in the wait queue. The following steps are taken:

1. Let \( \text{buffer} \) be \text{ValidateIntegerTypedArray}(\text{typedArray}, \text{true}).
2. Let \( \text{indexedPosition} \) be \text{ValidateAtomicAccess}(\text{typedArray}, \text{index}).
3. If \( \text{count} \) is \text{undefined}, let \( c \) be \(+\infty\).
4. Else,
   a. Let \( \text{intCount} \) be \text{ToIntegerOrInfinity}(\text{count}).
   b. Let \( c \) be \max(\text{intCount}, 0).
5. Let \( \text{block} \) be \text{buffer}[[\text{ArrayBufferData}}].
6. Let \( \text{arrayTypeName} \) be \text{typedArray}[[\text{TypedArrayName}]].
7. If \text{IsSharedArrayBuffer}(\text{buffer}) \) is \text{false}, return \(+0_{\text{V}}\).
8. Let \( WL \) be \text{GetWaiterList}(\text{block}, \text{indexedPosition}).
9. Let \( n \) be 0.
11. Let \( S \) be \text{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 \( S \).
   d. Set \( n \) to \( n + 1 \).
14. Return \( n \).

25.4.13 \text{Atomics.xor} ( \text{typedArray, index, value} )

The following steps are taken:

1. Let \( \text{xor} \) 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 \( \text{ByteListBitwiseOp}(\land, xBytes, yBytes) \).
2. Return ? \text{AtomicReadModifyWrite}(\text{typedArray}, \text{index}, \text{value}, \text{xor}).

25.4.14 \text{Atomics [ @@toStringTag ]}

The initial value of the \text{@@toStringTag} property is the String value "Atoms".
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 $2_\mathbb{F}$.

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(I[I])`.
          2. Let `newElement` be `InternalizeJSONProperty(val, *prop*, *reviver)`.  
          3. If `newElement` is *undefined*, then  
          4. Else,  
      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  
          3. Else,  
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 \texttt{Type}(\texttt{space}) is \texttt{Number}, then
   a. Let \texttt{spaceMV} be \texttt{! ToIntegerOrInfinity}(\texttt{space}).
   b. Set \texttt{spaceMV} to \texttt{min}(10, \texttt{spaceMV}).
   c. If \texttt{spaceMV} < 1, let \texttt{gap} be the empty String; otherwise let \texttt{gap} be the String value containing \texttt{spaceMV} occurrences of the code unit 0x0020 (SPACE).
7. Else if \texttt{Type}(\texttt{space}) is \texttt{String}, then
   a. If the length of \texttt{space} is 10 or less, let \texttt{gap} be \texttt{space}; otherwise let \texttt{gap} be the substring of \texttt{space} from 0 to 10.
8. Else,
   a. Let \texttt{gap} be the empty String.
9. Let \texttt{wrapper} be \texttt{! OrdinaryObjectCreate}(\%Object.prototype\%).
10. Perform \texttt{! CreateDataPropertyOrThrow}(\texttt{wrapper}, the empty String, \texttt{value}).
11. Let \texttt{state} be the Record \{ [[\texttt{ReplacerFunction}]]: \texttt{ReplacerFunction}, [[\texttt{Stack}]]: \texttt{stack}, [[\texttt{Indent}]]: \texttt{indent}, [[\texttt{Gap}]]: \texttt{gap}, [[\texttt{PropertyList}]]: \texttt{PropertyList}\}.
12. Return \texttt{? SerializeJSONProperty}(\texttt{state}, the empty String, \texttt{wrapper}).

The "length" property of the \texttt{stringify} function is 3._}

\textbf{NOTE 1} JSON structures are allowed to be nested to any depth, but they must be acyclic. If \texttt{value} is or contains a cyclic structure, then the \texttt{stringify} function must throw a \texttt{TypeError} exception. This is an example of a value that cannot be stringified:

\begin{verbatim}
   a = []; a[0] = a; my_text = JSON.stringify(a); // This must throw a TypeError.
\end{verbatim}

\textbf{NOTE 2} Symbolic primitive values are rendered as follows:

- The \texttt{null} value is rendered in JSON text as the String "null".
- The \texttt{undefined} value is not rendered.
- The \texttt{true} value is rendered in JSON text as the String "true".
- The \texttt{false} value is rendered in JSON text as the String "false".

\textbf{NOTE 3} String values are wrapped in QUOTATION MARK ("\) code units. The code units " and \ are escaped with \ prefix. 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).

\textbf{NOTE 4} Finite numbers are stringified as if by calling \texttt{ToString}(\texttt{number}). \texttt{NaN} and \texttt{Infinity} regardless of sign are represented as the String "null".

\textbf{NOTE 5} Values that do not have a JSON representation (such as \texttt{undefined} and functions) do not produce a String. Instead they produce the \texttt{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.
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 \( \text{StringToCodePoints}(value) \), 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 \( \text{UnicodeEscape}(unit) \).
   c. Else,
      i. Set \( product \) to the string-concatenation of \( product \) and \( \text{UTF16EncodeCodePoint}(C) \).

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

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Unicode Character Name</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0008</td>
<td>BACKSPACE</td>
<td>\b</td>
</tr>
<tr>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>\t</td>
</tr>
<tr>
<td>U+000A</td>
<td>LINE FEED (LF)</td>
<td>\n</td>
</tr>
<tr>
<td>U+000C</td>
<td>FORM FEED (FF)</td>
<td>\f</td>
</tr>
<tr>
<td>U+000D</td>
<td>CARRIAGE RETURN (CR)</td>
<td>\r</td>
</tr>
<tr>
<td>U+0022</td>
<td>QUOTATION MARK</td>
<td>&quot;</td>
</tr>
<tr>
<td>U+005C</td>
<td>REVERSE SOLIDUS</td>
<td>\</td>
</tr>
</tbody>
</table>

#### 25.5.2.3 \( \text{UnicodeEscape}(C) \)

The abstract operation \( \text{UnicodeEscape}(C) \) 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 \( \text{SerializeJSONObject}(state, value) \)

The abstract operation \( \text{SerializeJSONObject}(state, value) \) takes arguments \( state \) and \( value \). It serializes an object. It performs the following steps when called:

1. If \( state.[[\text{Stack}]] \) contains \( value \), throw a \text{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).
          ii. 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.

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 \( \text{index} \) be 0.

8. Repeat, while \( \text{index} < \text{len} \),
   a. Let \( \text{strP} \) be ? SerializeJSONProperty(\( \text{state} \), \! ToString(\( \text{F}(\text{index}) \)), \( \text{value} \)).
   b. If \( \text{strP} \) is \text{undefined}, then
      i. Append "null" to \( \text{partial} \).
   c. Else,
      i. Append \( \text{strP} \) to \( \text{partial} \).
   d. Set \( \text{index} \) to \( \text{index} + 1 \).

9. If \( \text{partial} \) is empty, then
   a. Let \( \text{final} \) be "[]".

10. Else,
    a. If \( \text{state}.[[\text{Gap}]] \) is the empty String, then
       i. Let \( \text{properties} \) be the String value formed by concatenating all the element Strings of \( \text{partial} \) 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 \( \text{final} \) be the string-concatenation of "[", \( \text{properties} \), and "]".
    b. Else,
       i. Let \( \text{separator} \) be the string-concatenation of the code unit 0x002C (COMMA), the code unit 0x000A (LINE FEED), and \( \text{state}.[[\text{Indent}]] \).
       ii. Let \( \text{properties} \) be the String value formed by concatenating all the element Strings of \( \text{partial} \) with each adjacent pair of Strings separated with \( \text{separator} \). The \( \text{separator} \) String is not inserted either before the first String or after the last String.
       iii. Let \( \text{final} \) be the string-concatenation of "[", the code unit 0x000A (LINE FEED), \( \text{state}.[[\text{Indent}]] \), \( \text{properties} \), the code unit 0x000A (LINE FEED), \( \text{stepback} \), and "]".

11. Remove the last element of \( \text{state}.[[\text{Stack}]] \).
12. Set \( \text{state}.[[\text{Indent}]] \) to \( \text{stepback} \).
13. Return \( \text{final} \).

NOTE

The representation of arrays includes only the elements between zero and \( \text{array.length} - 1 \) inclusive. Properties whose keys are not \text{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.
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.

```javascript
let 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.
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>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@iterator</td>
<td>A function that returns an Iterator object.</td>
<td>The returned object must conform to the Iterator interface.</td>
</tr>
</tbody>
</table>

27.1.2 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;next&quot;</td>
<td>A function that returns an <code>IteratorResult</code> object.</td>
<td>The returned object must conform to the <code>IteratorResult</code> interface. If a previous call to the <code>next</code> method of an <code>Iterator</code> has returned an <code>IteratorResult</code> object whose &quot;done&quot; property is <code>true</code>, then all subsequent calls to the <code>next</code> method of that object should also return an <code>IteratorResult</code> object whose &quot;done&quot; property is <code>true</code>. However, this requirement is not enforced.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;return&quot;</td>
<td>A function that returns an <code>IteratorResult</code> object.</td>
<td>The returned object must conform to the <code>IteratorResult</code> interface. Invoking this method notifies the <code>Iterator</code> object that the caller does not intend to make any more <code>next</code> method calls to the <code>Iterator</code>. The returned <code>IteratorResult</code> object will typically have a &quot;done&quot; property whose value is <code>true</code>, and a &quot;value&quot; property with the value passed as the argument of the <code>return</code> method. However, this requirement is not enforced.</td>
</tr>
<tr>
<td>&quot;throw&quot;</td>
<td>A function that returns an <code>IteratorResult</code> object.</td>
<td>The returned object must conform to the <code>IteratorResult</code> interface. Invoking this method notifies the <code>Iterator</code> 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 <code>throw</code> the value passed as the argument. If the method does not <code>throw</code>, the returned <code>IteratorResult</code> object will typically have a &quot;done&quot; property whose value is <code>true</code>.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@asyncIterator</td>
<td>A function that returns an AsyncIterator object.</td>
<td>The returned object must conform to the AsyncIterator interface.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;next&quot;</td>
<td>A function that returns a promise for an IteratorResult object.</td>
<td>The returned promise, when fulfilled, must fulfill with an object which conforms to the IteratorResult interface. If a previous call to the next method of an AsyncIterator has returned a promise for an IteratorResult object whose &quot;done&quot; property is true, then all subsequent calls to the next method of that object should also return a promise for an IteratorResult object whose &quot;done&quot; property is true. However, this requirement is not enforced. Additionally, the IteratorResult object that serves as a fulfillment value should have a &quot;value&quot; property whose value is not a promise (or &quot;thenable&quot;). However, this requirement is also not enforced.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;return&quot;</td>
<td>A function that returns a promise for an IteratorResult object.</td>
<td>The returned promise, when fulfilled, must fulfill with an object which conforms to the IteratorResult interface. Invoking this method notifies the AsyncIterator object that the caller does not intend to make any more next method calls to the AsyncIterator. The returned promise will fulfill with an IteratorResult object which will typically have a &quot;done&quot; property whose value is true, and a &quot;value&quot; property with the value passed as the argument of the return method. However, this requirement is not enforced. Additionally, the IteratorResult object that serves as a fulfillment value should have a &quot;value&quot; property whose value is not a promise (or &quot;thenable&quot;). 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 &quot;value&quot; property of the returned promise's IteratorResult object fulfillment value. However, these requirements are also not enforced.</td>
</tr>
<tr>
<td>&quot;throw&quot;</td>
<td>A function that returns a promise for an IteratorResult object.</td>
<td>The returned promise, when fulfilled, must fulfill with an object which conforms to the IteratorResult interface. Invoking this method notifies the AsyncIterator 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. If the returned promise is fulfilled, the IteratorResult fulfillment value will typically have a &quot;done&quot; property whose value is true. Additionally, it should have a &quot;value&quot; property whose value is not a promise (or &quot;thenable&quot;), but this requirement is not enforced.</td>
</tr>
</tbody>
</table>

#### 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;done&quot;</td>
<td>Either true or false.</td>
<td>This is the result status of an iterator <code>next</code> method call. If the end of the iterator was reached &quot;done&quot; is true. If the end was not reached &quot;done&quot; is false and a value is available. If a &quot;done&quot; property (either own or inherited) does not exist, it is considered to have the value false.</td>
</tr>
<tr>
<td>&quot;value&quot;</td>
<td>Any ECMAScript language value.</td>
<td>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, &quot;value&quot; is undefined. In that case, the &quot;value&quot; property may be absent from the conforming object if it does not inherit an explicit &quot;value&quot; property.</td>
</tr>
</tbody>
</table>

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:

```javascript
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 $\text{IteratorNext}(\text{syncIteratorRecord})$.
7. IfAbruptRejectPromise(result, promiseCapability).
8. Return $\text{AsyncFromSyncIteratorContinuation}(\text{result}, \text{promiseCapability})$.

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 $\text{NewPromiseCapability}(%\text{Promise}%)$.
4. Let syncIterator be $O$.[[SyncIteratorRecord]].[[Iterator]].
5. Let return be $\text{GetMethod}(\text{syncIterator}, \"\text{return}\")$.
6. IfAbruptRejectPromise(return, promiseCapability).
7. If return is undefined, then
   a. Let iterResult be $\text{CreateIterResultObject}(\text{value}, \text{true})$.
   b. Perform $\text{Call}(\text{promiseCapability}.[[\text{Resolve}]], \text{undefined, \"iterResult \")$.
   c. Return promiseCapability.[[Promise]].
8. If value is present, then
   a. Let result be $\text{Call}(\text{return}, \text{syncIterator}, \{"\text{value}\})$.
9. Else,
   a. Let result be $\text{Call}(\text{return}, \text{syncIterator})$.
10. IfAbruptRejectPromise(result, promiseCapability).
11. If Type(result) is not Object, then
    a. Perform $\text{Call}(\text{promiseCapability}.[[\text{Reject}]], \text{undefined, \"a newly created TypeError object \")$.
    b. Return promiseCapability.[[Promise]].
12. Return $\text{AsyncFromSyncIteratorContinuation}(\text{result}, \text{promiseCapability})$.

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 $\text{NewPromiseCapability}(%\text{Promise}%)$.
4. Let syncIterator be $O$.[[SyncIteratorRecord]].[[Iterator]].
5. Let throw be $\text{GetMethod}(\text{syncIterator}, \"\text{throw}\")$.
6. IfAbruptRejectPromise(throw, promiseCapability).
7. If throw is undefined, then
   a. Perform $\text{Call}(\text{promiseCapability}.[[\text{Reject}]], \text{undefined, \"value \")$.
   b. Return promiseCapability.[[Promise]].
8. If value is present, then
   a. Let result be $\text{Call}(\text{throw}, \text{syncIterator}, \{"\text{value}\})$.
9. Else,
   a. Let result be $\text{Call}(\text{throw}, \text{syncIterator})$.
10. IfAbruptRejectPromise(result, promiseCapability).
11. If Type(result) is not Object, then
    a. Perform $\text{Call}(\text{promiseCapability}.[[\text{Reject}]], \text{undefined, \"a newly created TypeError object \")$.
    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>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[SyncIteratorRecord]]</code></td>
<td>A <code>Record</code>, of the type returned by <code>GetIterator</code>, representing the original synchronous iterator which is being adapted.</td>
</tr>
</tbody>
</table>

### 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)
Any Promise object is in one of three mutually exclusive states: fulfilled, rejected, and pending:

- A promise \( p \) is fulfilled if \( p.\text{then}(f, r) \) will immediately enqueue a Job to call the function \( f \).
- A promise \( p \) is rejected if \( p.\text{then}(f, r) \) 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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Promise]]</td>
<td>An object</td>
<td>An object that is usable as a promise.</td>
</tr>
<tr>
<td>[[Resolve]]</td>
<td>A function object</td>
<td>The function that is used to resolve the given promise object.</td>
</tr>
<tr>
<td>[[Reject]]</td>
<td>A function object</td>
<td>The function that is used to reject the given promise object.</td>
</tr>
</tbody>
</table>

#### 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
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.

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`.

**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`.

**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`.
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 $2^\pi$.

### 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**

- 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.
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` }.

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).
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.
   l. 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`.
b. Set `onFulfilled`[[Capability]] to `resultCapability`.
c. Set `onFulfilled`[[RemainingElements]] to `remainingElementsCount`.
d. Set `remainingElementsCount`[[Value]] to `remainingElementsCount`[[Value]] + 1.
e. Perform `Invoke(nextPromise, "then", « onFulfilled, resultCapability.[[Reject]] »)`.
f. 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 `1F`.

### 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)`.
NOTE

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`.

766
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`.

### 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]]`.

767
7. Let `promiseCapability` be `F.[Capability]`.
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`.
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.

### 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.

l. 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.\[\[Errors\]\]`.
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 `AgggregateError` object.
b. Perform `! DefinePropertyOrThrow(error, "errors", PropertyDescriptor { [[Configurable]]: true, [[Enumerable]]: false, [[Writable]]: true, [[Value]]: ! CreateArrayFromList(errors) })`.

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.

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.
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.
5. 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]],
d. Set \( \text{thenFinally} \)\( .[[\text{Constructor}]] \) to \( C \).

e. Set \( \text{thenFinally} \)\( .[[\text{OnFinally}}] \) to \( \text{onFinally} \).

f. Let \( \text{stepsCatchFinally} \) be the algorithm steps defined in \( \text{Catch Finally Functions} \).

g. Let \( \text{lengthCatchFinally} \) be the number of non-optional parameters of the function definition in \( \text{Catch Finally Functions} \).

h. Let \( \text{catchFinally} \) be \( ! \text{CreateBuiltinFunction}(\text{stepsCatchFinally}, \text{lengthCatchFinally}, \"\"", \("[[\text{Constructor}], [[\text{OnFinally}}]]) \).

i. Set \( \text{catchFinally} \)\( .[[\text{Constructor}]] \) to \( C \).

j. Set \( \text{catchFinally} \)\( .[[\text{OnFinally}}] \) to \( \text{onFinally} \).

7. Return ? Invoke(promise, "then", \("\text{thenFinally}, \text{catchFinally} \)\).
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.[[PromisesIsHandled]] is false, perform HostPromiseRejectionTracker(promise, "handle").
    d. Let rejectJob be NewPromiseReactionJob(rejectReaction, reason).
    e. Perform HostEnqueuePromiseJob(rejectJob.[[Job]], rejectJob.[[Realm]]).
12. Set `promise.[[PromisesHandled]]` 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

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[PromiseState]]</code></td>
<td>One of <code>pending</code>, <code>fulfilled</code>, or <code>rejected</code>. Governs how a promise will react to incoming calls to its <code>then</code> method.</td>
</tr>
<tr>
<td><code>[[PromiseResult]]</code></td>
<td>The value with which the promise has been fulfilled or rejected, if any. Only meaningful if <code>[[PromiseState]]</code> is not <code>pending</code>.</td>
</tr>
<tr>
<td><code>[[PromiseFulfillReactions]]</code></td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the <code>pending</code> state to the <code>fulfilled</code> state.</td>
</tr>
<tr>
<td><code>[[PromiseRejectReactions]]</code></td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the <code>pending</code> state to the <code>rejected</code> state.</td>
</tr>
<tr>
<td><code>[[PromisesHandled]]</code></td>
<td>A boolean indicating whether the promise has ever had a fulfillment or rejection handler; used in unhandled rejection tracking.</td>
</tr>
</tbody>
</table>

### 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.
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]]`.

#### 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]].

```
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 }. 

779
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

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[GeneratorState]]</code></td>
<td>The current execution state of the generator. The possible values are: <code>undefined</code>, <code>suspendedStart</code>, <code>suspendedYield</code>, <code>executing</code>, and <code>completed</code>.</td>
</tr>
<tr>
<td><code>[[GeneratorContext]]</code></td>
<td>The execution context that is used when executing the code of this generator.</td>
</tr>
<tr>
<td><code>[[GeneratorBrand]]</code></td>
<td>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>.</td>
</tr>
</tbody>
</table>

### 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 \textit{genContext} to \textit{generator}.

4. Set the code evaluation state of \textit{genContext} such that when evaluation is resumed for that execution context the following steps will be performed:
   a. If \textit{generatorBody} is a Parse Node, then
      i. Let \textit{result} be the result of evaluating \textit{generatorBody}.
   b. Else,
      i. \textbf{Assert:} \textit{generatorBody} is an Abstract Closure with no parameters.
      ii. Let \textit{result} be \textit{generatorBody}().
   c. \textbf{Assert:} If we return here, the generator either threw an exception or performed either an implicit or explicit return.
   d. Remove \textit{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 \textit{generator}.[[GeneratorState]] to \textit{completed}.
   f. Once a generator enters the \textit{completed} state it never leaves it and its associated execution context is never resumed. Any execution state associated with \textit{generator} can be discarded at this point.
   g. If \textit{result}.[[Type]] is \textit{normal}, let \textit{resultValue} be \textit{undefined}.
   h. Else if \textit{result}.[[Type]] is \textit{return}, let \textit{resultValue} be \textit{result}.[[Value]].
   i. Else,
      i. \textbf{Assert:} \textit{result}.[[Type]] is \textit{throw}.
      ii. Return \textbf{Completion}({\textit{result}}).
   j. Return \textbf{CreateIterResultObject}({\textit{resultValue}, true}).

5. Set \textit{generator}.[[GeneratorContext]] to \textit{genContext}.

6. Set \textit{generator}.[[GeneratorState]] to \textit{suspendedStart}.

7. Return \textbf{NormalCompletion}({\textit{undefined}}).

\subsection*{27.5.3.2 GeneratorValidate (generator, generatorBrand)}

The abstract operation GeneratorValidate takes arguments \textit{generator} and \textit{generatorBrand}. It performs the following steps when called:

1. Perform \textbf{? RequireInternalSlot}({\textit{generator}, [[GeneratorState]]}).
2. Perform \textbf{? RequireInternalSlot}({\textit{generator}, [[GeneratorBrand]]}).
3. If \textit{generator}.[[GeneratorBrand]] is not the same value as \textit{generatorBrand}, throw a \textbf{TypeError} exception.
4. \textbf{Assert:} \textit{generator} also has a [[GeneratorContext]] internal slot.
5. Let \textit{state} be \textit{generator}.[[GeneratorState]].
6. If \textit{state} is \textit{executing}, throw a \textbf{TypeError} exception.
7. Return \textit{state}.

\subsection*{27.5.3.3 GeneratorResume (generator, value, generatorBrand)}

The abstract operation GeneratorResume takes arguments \textit{generator}, \textit{value}, and \textit{generatorBrand}. It performs the following steps when called:

1. Let \textit{state} be \textbf{? GeneratorValidate}({\textit{generator}, \textit{generatorBrand}}).
2. If \textit{state} is \textit{completed}, return \textbf{CreateIterResultObject}({\textit{undefined}, true}).
3. \textbf{Assert:} \textit{state} is either \textit{suspendedStart} or \textit{suspendedYield}.
4. Let \textit{genContext} be \textit{generator}.[[GeneratorContext]].
5. Let \textit{methodContext} be the running execution context.
6. Suspend \textit{methodContext}. \\
7. Set \textit{generator}.[[GeneratorState]] to executing. \\
8. Push \textit{genContext} onto the execution context stack; \textit{genContext} is now the running execution context. \\
9. Resume the suspended evaluation of \textit{genContext} using \texttt{NormalCompletion(\textit{value})} as the result of the operation that suspended it. Let \textit{result} be the value returned by the resumed computation. \\
10. \textbf{Assert:} When we return here, \textit{genContext} has already been removed from the execution context stack and \textit{methodContext} is the currently running execution context. \\
11. Return \texttt{Completion(\textit{result})}.

\subsection{27.5.3.4 GeneratorResumeAbrupt (generator, abruptCompletion, generatorBrand) }

The abstract operation \texttt{GeneratorResumeAbrupt} takes arguments \textit{generator}, \textit{abruptCompletion} (a \texttt{Completion Record} whose [[Type]] is \texttt{return} or \texttt{throw}), and \textit{generatorBrand}. It performs the following steps when called:

1. Let \textit{state} be ? \texttt{GeneratorValidate(generator, generatorBrand)}. \\
2. If \textit{state} is \texttt{suspendedStart}, then  
   a. Set \textit{generator}.[[GeneratorState]] to \texttt{completed}. 
   b. Once a generator enters the \texttt{completed} state it never leaves it and its associated execution context is never resumed. Any execution state associated with \textit{generator} can be discarded at this point. 
   c. Set \textit{state} to \texttt{completed}. \\
3. If \textit{state} is \texttt{completed}, then  
   a. If \textit{abruptCompletion}.[[Type]] is \texttt{return}, then  
      i. Return \texttt{CreateIterResultObject(abruptCompletion}.[[Value]], true). 
   b. Return \texttt{Completion(abruptCompletion)}. \\
4. \textbf{Assert:} \textit{state} is \texttt{suspendedYield}. \\
5. Let \textit{genContext} be \textit{generator}.[[GeneratorContext]]. \\
6. Let \textit{methodContext} be the running execution context. \\
7. Suspend \textit{methodContext}. \\
8. Set \textit{generator}.[[GeneratorState]] to executing. \\
9. Push \textit{genContext} onto the execution context stack; \textit{genContext} is now the running execution context. \\
10. Resume the suspended evaluation of \textit{genContext} using \textit{abruptCompletion} as the result of the operation that suspended it. Let \textit{result} be the completion record returned by the resumed computation. \\
11. \textbf{Assert:} When we return here, \textit{genContext} has already been removed from the execution context stack and \textit{methodContext} is the currently running execution context. \\
12. Return \texttt{Completion(result)}.

\subsection{27.5.3.5 GetGeneratorKind ()}

The abstract operation \texttt{GetGeneratorKind} takes no arguments. It performs the following steps when called:

1. Let \textit{genContext} be the running execution context. \\
2. If \textit{genContext} does not have a Generator component, return \texttt{non-generator}. \\
3. Let \textit{generator} be the Generator component of \textit{genContext}. \\
4. If \textit{generator} has an [[AsyncGeneratorState]] internal slot, return \texttt{async}. \\
5. Else, return \texttt{sync}.

\subsection{27.5.3.6 GeneratorYield (iterNextObj) }

\subsection{27.5.3.7 GeneratorSet (genContext, normalCompletion) }

The abstract operation \texttt{GeneratorSet} takes arguments \textit{genContext} and \texttt{normalCompletion} (a Completion Record). It performs the following steps when called:

1. Let \textit{result} be the value returned by \texttt{normalCompletion}. \\
2. If \textit{result} is \texttt{throw}, then  
   a. Set \textit{generator}.[[GeneratorState]] to \texttt{completed}. 
   b. Once a generator enters the \texttt{completed} state it never leaves it and its associated execution context is never resumed. Any execution state associated with \textit{generator} can be discarded at this point. 
   c. Set \textit{state} to \texttt{completed}. 
3. If \textit{state} is \texttt{completed}, then  
   a. If \textit{abruptCompletion}.[[Type]] is \texttt{return}, then  
      i. Return \texttt{CreateIterResultObject(abruptCompletion}.[[Value]], true). 
   b. Return \texttt{Completion(abruptCompletion)}. \\
4. \textbf{Assert:} \textit{state} is \texttt{suspendedYield}. \\
5. Let \textit{genContext} be \textit{generator}.[[GeneratorContext]]. \\
6. Let \textit{methodContext} be the running execution context. \\
7. Suspend \textit{methodContext}. \\
8. Set \textit{generator}.[[GeneratorState]] to executing. \\
9. Push \textit{genContext} onto the execution context stack; \textit{genContext} is now the running execution context. \\
10. Resume the suspended evaluation of \textit{genContext} using \textit{abruptCompletion} as the result of the operation that suspended it. Let \textit{result} be the completion record returned by the resumed computation. \\
11. \textbf{Assert:} When we return here, \textit{genContext} has already been removed from the execution context stack and \textit{methodContext} is the currently running execution context. \\
12. Return \texttt{Completion(result)}.
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` 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` to `generatorBrand`.
5. Set `generator` 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).

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 }.

27.6.1.4 AsyncGenerator.prototype.throw (exception)

1. Let generator be the this value.
2. Let completion be ThrowCompletion(exception).

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

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[AsyncGeneratorState]]</td>
<td>The current execution state of the async generator. The possible values are: undefined, suspendedStart, suspendedYield, executing, awaiting-return, and completed.</td>
</tr>
<tr>
<td>[[AsyncGeneratorContext]]</td>
<td>The execution context that is used when executing the code of this async generator.</td>
</tr>
<tr>
<td>[[AsyncGeneratorQueue]]</td>
<td>A List of AsyncGeneratorRequest records which represent requests to resume the async generator.</td>
</tr>
<tr>
<td>[[GeneratorBrand]]</td>
<td>A brand used to distinguish different kinds of async generators. The [[GeneratorBrand]] of async generators declared by ECMAScript source text is always empty.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Completion]]</td>
<td>A Completion record</td>
<td>The completion which should be used to resume the async generator.</td>
</tr>
<tr>
<td>[[Capability]]</td>
<td>A PromiseCapability Record</td>
<td>The promise capabilities associated with this request.</td>
</tr>
</tbody>
</table>

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 \( \text{result} \) be \( \text{generatorBody}() \).

c. **Assert**: If we return here, the async generator either threw an exception or performed either an implicit or explicit return.

d. Remove \( \text{genContext} \) 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 \( \text{generator}[[\text{AsyncGeneratorState}]] \) to \( \text{completed} \).

f. If \( \text{result} \) is a normal completion, let \( \text{resultValue} \) be \( \text{undefined} \).

g. Else,
   i. Let \( \text{resultValue} \) be \( \text{result}[[\text{Value}]] \).
   ii. If \( \text{result}[[\text{Type}]] \) is not \( \text{return} \), then
      1. Return \(! \text{AsyncGeneratorReject}(\text{generator}, \text{resultValue})\).

h. Return \(! \text{AsyncGeneratorResolve}(\text{generator}, \text{resultValue}, \text{true})\).

6. Set \( \text{generator}[[\text{AsyncGeneratorContext}]] \) to \( \text{genContext} \).

7. Set \( \text{generator}[[\text{AsyncGeneratorState}]] \) to \( \text{suspendedStart} \).

8. Set \( \text{generator}[[\text{AsyncGeneratorQueue}]] \) to a new empty List.

9. Return \( \text{undefined} \).

### 27.6.3.3 AsyncGeneratorValidate (\( \text{generator, generatorBrand} \))

The abstract operation AsyncGeneratorValidate takes arguments \( \text{generator} \) and \( \text{generatorBrand} \). It performs the following steps when called:

1. Perform \(! \text{RequireInternalSlot}(\text{generator}, [[\text{AsyncGeneratorContext}]]))\).

2. Perform \(! \text{RequireInternalSlot}(\text{generator}, [[\text{AsyncGeneratorState}]]))\).

3. Perform \(! \text{RequireInternalSlot}(\text{generator}, [[\text{AsyncGeneratorQueue}]]))\).

4. If \( \text{generator}[[\text{GeneratorBrand}]] \) is not the same value as \( \text{generatorBrand} \), throw a \text{TypeError} exception.

### 27.6.3.4 AsyncGeneratorResolve (\( \text{generator, value, done} \))

The abstract operation AsyncGeneratorResolve takes arguments \( \text{generator, value, and done} \) (a Boolean). It performs the following steps when called:

1. **Assert**: \( \text{generator} \) is an AsyncGenerator instance.

2. Let \( \text{queue} \) be \( \text{generator}[[\text{AsyncGeneratorQueue}]] \).

3. **Assert**: \( \text{queue} \) is not an empty List.

4. Let \( \text{next} \) be the first element of \( \text{queue} \).

5. Remove the first element from \( \text{queue} \).

6. Let \( \text{promiseCapability} \) be \( \text{next}[[\text{Capability}]] \).

7. Let \( \text{iteratorResult} \) be \(! \text{CreateIterResultObject}(\text{value, done})\).

8. Perform \(! \text{Call}(\text{promiseCapability}[[\text{Resolve}]], \text{undefined}, \langle \text{iteratorResult} \rangle)\).

9. Perform \(! \text{AsyncGeneratorResumeNext}(\text{generator})\).

10. Return \( \text{undefined} \).

### 27.6.3.5 AsyncGeneratorReject (\( \text{generator, exception} \))

The abstract operation AsyncGeneratorReject takes arguments \( \text{generator} \) and \( \text{exception} \). It performs the following steps when called:

1. **Assert**: \( \text{generator} \) 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 »).
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.


### 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.
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]],...
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]]`.

**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
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]] »).
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).

### 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.

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.

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Order]]</td>
<td>SeqCst</td>
<td>The weakest ordering guaranteed by the memory model for the event.</td>
</tr>
<tr>
<td></td>
<td>Unordered</td>
<td></td>
</tr>
<tr>
<td>[[NoTear]]</td>
<td>A Boolean</td>
<td>Whether this event is allowed to read from multiple write events on equal range as this event.</td>
</tr>
<tr>
<td>[[Block]]</td>
<td>A Shared Data Block</td>
<td>The block the event operates on.</td>
</tr>
<tr>
<td>[[ByteIndex]]</td>
<td>A non-negative integer</td>
<td>The byte address of the read in [[Block]].</td>
</tr>
<tr>
<td>[[ElementSize]]</td>
<td>A non-negative integer</td>
<td>The size of the read.</td>
</tr>
</tbody>
</table>

### Table 77: WriteSharedMemory Event Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Order]]</td>
<td>SeqCst</td>
<td>The weakest ordering guaranteed by the memory model for the event.</td>
</tr>
<tr>
<td></td>
<td>Unordered</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Init</td>
<td></td>
</tr>
<tr>
<td>[[NoTear]]</td>
<td>A Boolean</td>
<td>Whether this event is allowed to be read from multiple read events with equal range as this event.</td>
</tr>
<tr>
<td>[[Block]]</td>
<td>A Shared Data Block</td>
<td>The block the event operates on.</td>
</tr>
<tr>
<td>[[ByteIndex]]</td>
<td>A non-negative integer</td>
<td>The byte address of the write in [[Block]].</td>
</tr>
<tr>
<td>[[ElementSize]]</td>
<td>A non-negative integer</td>
<td>The size of the write.</td>
</tr>
<tr>
<td>[[Payload]]</td>
<td>A List</td>
<td>The List of byte values to be read by other events.</td>
</tr>
</tbody>
</table>
Table 78: ReadModifyWriteSharedMemory Event Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Order]]</td>
<td>SeqCst</td>
<td>Read-modify-write events are always sequentially consistent.</td>
</tr>
<tr>
<td>[[NoTear]]</td>
<td>true</td>
<td>Read-modify-write events cannot tear.</td>
</tr>
<tr>
<td>[[Block]]</td>
<td>A Shared Data Block</td>
<td>The block the event operates on.</td>
</tr>
<tr>
<td>[[ByteIndex]]</td>
<td>A non-negative integer</td>
<td>The byte address of the read-modify-write in [[Block]].</td>
</tr>
<tr>
<td>[[ElementSize]]</td>
<td>A non-negative integer</td>
<td>The size of the read-modify-write.</td>
</tr>
<tr>
<td>[[Payload]]</td>
<td>A List</td>
<td>The List of byte values to be passed to [[ModifyOp]].</td>
</tr>
<tr>
<td>[[ModifyOp]]</td>
<td>A read-modify-write</td>
<td>An abstract closure that returns a modified List of byte values from a</td>
</tr>
<tr>
<td></td>
<td>modification function</td>
<td>read List of byte values and [[Payload]].</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[AgentSignifier]]</td>
<td>A value that admits equality testing</td>
<td>The agent whose evaluation resulted in this ordering.</td>
</tr>
<tr>
<td>[[EventList]]</td>
<td>A List of events</td>
<td>Events are appended to the list during evaluation.</td>
</tr>
<tr>
<td>[[AgentSynchronizesWith]]</td>
<td>A List of pairs of Synchronize events</td>
<td>Synchronize relationships introduced by the operational semantics.</td>
</tr>
</tbody>
</table>

### 29.3 Chosen Value Records

A *Chosen Value Record* is a **Record** with the following fields.

Table 80: **Chosen Value Record** Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Event]]</td>
<td>A Shared Data Block event</td>
<td>The ReadSharedMemory or ReadModifyWriteSharedMemory event that was introduced for this chosen value.</td>
</tr>
<tr>
<td>[[ChosenValue]]</td>
<td>A List of byte values</td>
<td>The bytes that were nondeterministically chosen during evaluation.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[EventsRecords]]</td>
<td>A List of Agent Events Records.</td>
<td>Maps an agent to Lists of events appended during the evaluation.</td>
</tr>
<tr>
<td>[[ChosenValues]]</td>
<td>A List of Chosen Value Records.</td>
<td>Maps ReadSharedMemory or ReadModifyWriteSharedMemory events to the List of byte values chosen during the evaluation.</td>
</tr>
<tr>
<td>[[AgentOrder]]</td>
<td>An agent-order Relation.</td>
<td>Defined below.</td>
</tr>
<tr>
<td>[[ReadsBytesFrom]]</td>
<td>A reads-bytes-from mathematical function.</td>
<td>Defined below.</td>
</tr>
<tr>
<td>[[ReadsFrom]]</td>
<td>A reads-from Relation.</td>
<td>Defined below.</td>
</tr>
<tr>
<td>[[HostSynchronizesWith]]</td>
<td>A host-synchronizes-with Relation.</td>
<td>Defined below.</td>
</tr>
<tr>
<td>[[HappensBefore]]</td>
<td>A happens-before Relation.</td>
<td>Defined below.</td>
</tr>
</tbody>
</table>

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.\([\text{EventsRecords}]\), do
   a. For each event \( E \) of \( aer.\([\text{EventList}]\), 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 $\text{EventSet}(\text{execution})$, do
   a. If $E$ is a $\text{ReadSharedMemory}$, $\text{WriteSharedMemory}$, or $\text{ReadModifyWriteSharedMemory}$ event, add $E$ to $\text{events}$.
3. Return $\text{events}$.

29.5.3 $\text{HostEventSet}(\text{execution})$

The abstract operation $\text{HostEventSet}$ takes argument $\text{execution}$ (a candidate execution). It performs the following steps when called:

1. Let $\text{events}$ be an empty Set.
2. For each event $E$ of $\text{EventSet}(\text{execution})$, do
   a. If $E$ is not in $\text{SharedDataBlockEventSet}(\text{execution})$, add $E$ to $\text{events}$.
3. Return $\text{events}$.

29.5.4 $\text{ComposeWriteEventBytes}(\text{execution}, \text{byteIndex}, \text{Ws})$

The abstract operation $\text{ComposeWriteEventBytes}$ takes arguments $\text{execution}$ (a candidate execution), $\text{byteIndex}$ (a non-negative integer), and $\text{Ws}$ (a List of $\text{WriteSharedMemory}$ or $\text{ReadModifyWriteSharedMemory}$ events). It performs the following steps when called:

1. Let $\text{byteLocation}$ be $\text{byteIndex}$.
2. Let $\text{bytesRead}$ be a new empty List.
3. For each element $W$ of $\text{Ws}$, do
   a. Assert: $W$ has $\text{byteLocation}$ in its range.
   b. Let $\text{payloadIndex}$ be $\text{byteLocation} - W.[[\text{ByteIndex}]]$.
   c. If $W$ is a $\text{WriteSharedMemory}$ event, then
      i. Let $\text{byte}$ be $W.[[\text{Payload}}][\text{payloadIndex}]$.
   d. Else,
      i. Assert: $W$ is a $\text{ReadModifyWriteSharedMemory}$ event.
      ii. Let $\text{bytes}$ be $\text{ValueOfReadEvent}(\text{execution}, W)$.
      iii. Let $\text{bytesModified}$ be $W.[[\text{ModifyOp}}](\text{bytes, W.[[Payload}]})$.
      iv. Let $\text{byte}$ be $\text{bytesModified}[\text{payloadIndex}]$.
   e. Append $\text{byte}$ to $\text{bytesRead}$.
   f. Set $\text{byteLocation}$ to $\text{byteLocation} + 1$.
4. Return $\text{bytesRead}$.

NOTE 1
The read-modify-write modification $[[\text{ModifyOp}}]$ is given by the function properties on the Atomics object that introduce $\text{ReadModifyWriteSharedMemory}$ 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 $\text{ReadSharedMemory}$ and $\text{ReadModifyWriteSharedMemory}$ events.

29.5.5 $\text{ValueOfReadEvent}(\text{execution}, R)$

The abstract operation $\text{ValueOfReadEvent}$ takes arguments $\text{execution}$ (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 $\text{execution}.\{\text{ReadsBytesFrom}\}(R)$.
3. Assert: $Ws$ is a List of WriteSharedMemory or ReadModifyWriteSharedMemory events with length equal to $R$.\{ElementSize\].
4. Return $\text{ComposeWriteEventBytes}(\text{execution}, R.\{\text{ByteIndex}\}, Ws)$.

# 29.6 Relations of Candidate Executions

## 29.6.1 agent-order

For a candidate execution $\text{execution}$, $\text{execution}.\{\text{AgentOrder}\}$ is a Relation on events that satisfies the following.

- For each pair $(E, D)$ in $\text{EventSet}(\text{execution})$, $(E, D)$ is in $\text{execution}.\{\text{AgentOrder}\}$ if there is some Agent Events Record $\text{aer}$ in $\text{execution}.\{\text{EventsRecords}\}$ such that $E$ and $D$ are in $\text{aer}.\{\text{EventList}\}$ and $E$ is before $D$ in List order of $\text{aer}.\{\text{EventList}\}$.

**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 $\text{execution}$, $\text{execution}.\{\text{ReadsBytesFrom}\}$ is a mathematical function mapping events in $\text{SharedDataBlockEventSet}(\text{execution})$ to Lists of events in $\text{SharedDataBlockEventSet}(\text{execution})$ that satisfies the following conditions.

- For each ReadSharedMemory or ReadModifyWriteSharedMemory event $R$ in $\text{SharedDataBlockEventSet}(\text{execution})$, $\text{execution}.\{\text{ReadsBytesFrom}\}(R)$ is a List of length $R.\{\text{ElementSize}\}$ 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 $R.\{\text{ByteIndex}\} + i$ in its range.
  - $R$ is not in $Ws$.

## 29.6.3 reads-from

For a candidate execution $\text{execution}$, $\text{execution}.\{\text{ReadsFrom}\}$ is the least Relation on events that satisfies the following.

- For each pair $(R, W)$ in $\text{SharedDataBlockEventSet}(\text{execution})$, $(R, W)$ is in $\text{execution}.\{\text{ReadsFrom}\}$ if $W$ is in $\text{execution}.\{\text{ReadsBytesFrom}\}(R)$.

## 29.6.4 host-synchronizes-with

For a candidate execution $\text{execution}$, $\text{execution}.\{\text{HostSynchronizesWith}\}$ is a host-provided strict partial order on host-specific events that satisfies at least the following.
- If \((E, D)\) is in \(\text{execution.}[[\text{HostSynchronizesWith}]]\), \(E\) and \(D\) are in \(\text{HostEventSet(}\text{execution}\text{)}\).
- There is no cycle in the union of \(\text{execution.}[[\text{HostSynchronizesWith}]]\) and \(\text{execution.}[[\text{AgentOrder}]]\).

**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 \text{postMessage} between HTML workers.

### 29.6.5 synchronizes-with

For a candidate execution \(\text{execution, execution.}[[\text{SynchronizesWith}]]\) is the least Relation on events that satisfies the following.

- For each pair \((R, W)\) in \(\text{execution.}[[\text{ReadsFrom}]]\), \((W, R)\) is in \(\text{execution.}[[\text{SynchronizesWith}]]\) if \(R.[[\text{Order}]]\) is \text{SeqCst}, \(W.[[\text{Order}]]\) is \text{SeqCst}, and \(R\) and \(W\) have equal ranges.
- For each element \(\text{eventsRecord of execution.}[[\text{EventsRecords}]]\), the following is true.
  - For each pair \((S, Sw)\) in \(\text{eventsRecord.}[[\text{AgentSynchronizesWith}]]\), \((S, Sw)\) is in \(\text{execution.}[[\text{SynchronizesWith}]]\).
  - For each pair \((E, D)\) in \(\text{execution.}[[\text{HostSynchronizesWith}]]\), \((E, D)\) is in \(\text{execution.}[[\text{SynchronizesWith}]]\).

**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 \text{SeqCst} events related by \text{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 \text{reads-from} other writes than \(W\).

### 29.6.6 happens-before

For a candidate execution \(\text{execution, execution.}[[\text{HappensBefore}]]\) is the least Relation on events that satisfies the following.

- For each pair \((E, D)\) in \(\text{execution.}[[\text{AgentOrder}]]\), \((E, D)\) is in \(\text{execution.}[[\text{HappensBefore}]]\).
- For each pair \((E, D)\) in \(\text{execution.}[[\text{SynchronizesWith}]]\), \((E, D)\) is in \(\text{execution.}[[\text{HappensBefore}]]\).
- For each pair \((E, D)\) in \(\text{SharedDataBlockEventSet(}\text{execution}\text{)}\), \((E, D)\) is in \(\text{execution.}[[\text{HappensBefore}]]\) if \(E.\[[\text{Order}]]\) is \text{Init} and \(E\) and \(D\) have overlapping ranges.
- For each pair \((E, D)\) in \(\text{EventSet(}\text{execution}\text{)}\), \((E, D)\) is in \(\text{execution.}[[\text{HappensBefore}]]\) if there is an event \(F\) such that the pairs \((E, F)\) and \((F, D)\) are in \(\text{execution.}[[\text{HappensBefore}]]\).
NOTE: 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
   1. Let `chosenValueRecord` be the element of `execution.[[ChosenValues]]` whose `[[Event]]` field is *R*.
   2. Let `chosenValue` be `chosenValueRecord.[[ChosenValue]]`.
   3. Let `readValue` be `ValueOfReadEvent(execution, *R*)`.
   4. Let `chosenLen` be the number of elements of `chosenValue`.
   5. Let `readLen` be the number of elements of `readValue`.
   6. If `chosenLen ≠ readLen`, then
      1. Return **false**.
   7. If `chosenValue[i] ≠ readValue[i]` for any integer value *i* in the range 0 through `chosenLen`, exclusive, then
      1. 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
   1. Let `Ws` be `execution.[[ReadsBytesFrom]](*R*)`.
   2. Let `byteLocation` be `*R.[[ByteIndex]]`.
   3. For each element *W* of `Ws`, do
      1. If (*R, *W*) is in `execution.[[HappensBefore]]`, then
         1. Return **false**.
      2. 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**.
      3. 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-prefixd 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
   $
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 ∉ 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 \f \n \r \v \t \b \f \n \r \v \t]
  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 ::
   RegularExpressionNon Terminator but not one of * or \ or / or [ 
   RegularExpressionBackslashSequence
   RegularExpressionClass

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

RegularExpressionBackslashSequence ::
   \ RegularExpressionNon Terminator

RegularExpressionNon Terminator ::
   SourceCharacter but not LineTerminator

RegularExpressionClass ::
   [ RegularExpressionClassChars ]

RegularExpressionClassChars ::
   [empty]
   RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar ::
   RegularExpressionNon Terminator but not one of ] or \ 
   RegularExpressionBackslashSequence

RegularExpressionFlags ::
   [empty]
   RegularExpressionFlags IdentifierPart

Template ::
   NoSubstitutionTemplate
   TemplateHead
A.2 Expressions

IdentifierReference[Yield, Await] :
  Identifier
  [-Yield]yield
  [-Await]await

BindingIdentifier[Yield, Await] :
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
the interpretation of CoverParenthesizedExpressionAndArrowParameterList is refined using the following grammar:

ParenthesizedExpression [Yield, Await] :
  ( Expression [+In, ?Yield, ?Await] )

Literal :
  NullLiteral
  BooleanLiteral
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]

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]
TemplateMiddleList: [Yield, Await, Tagged]:
  TemplateMiddleExpression: [+In, ?Yield, ?Await]
  TemplateMiddleList: [Yield, ?Await, ?Tagged]
TemplateMiddleList: [Yield, ?Await, ?Tagged]
TemplateMiddleList: [Yield, ?Await, ?Tagged]
  TemplateMiddle: Expression: [+In, ?Yield, ?Await]
  TemplateMiddleList: [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]
  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

\[\text{CallExpression}[\text{Yield, Await}] : \text{CoverCallExpressionAndAsyncArrowHead}[?\text{Yield}, ?\text{Await}]\]

the interpretation of \text{CoverCallExpressionAndAsyncArrowHead} is refined using the following grammar:

\[\text{CallMemberExpression}[\text{Yield, Await}] :
  \text{MemberExpression}[?\text{Yield}, ?\text{Await}] \text{ Arguments}[?\text{Yield, ?Await}]\]

\[\text{SuperCall}[\text{Yield, Await}] :
  \text{super Arguments}[?\text{Yield, ?Await}]\]

\[\text{ImportCall}[\text{Yield, Await}] :
  \text{import ( AssignmentExpression }[+\text{In, ?Yield, ?Await}] \text{ )}\]

\[\text{Arguments}[\text{Yield, Await}] :
  ( )
  ( \text{ArgumentList}[?\text{Yield, ?Await}] )
  ( \text{ArgumentList}[?\text{Yield, ?Await}], )\]

\[\text{ArgumentList}[\text{Yield, Await}] :
  \text{AssignmentExpression }[+\text{In, ?Yield, ?Await}]\]
  ... \text{AssignmentExpression }[+\text{In, ?Yield, ?Await}]
  \text{ArgumentList}[?\text{Yield, ?Await}], \text{AssignmentExpression }[+\text{In, ?Yield, ?Await}]
  \text{ArgumentList}[?\text{Yield, ?Await}], ..., \text{AssignmentExpression }[+\text{In, ?Yield, ?Await}]

\[\text{OptionalExpression}[\text{Yield, Await}] :
  \text{MemberExpression}[?\text{Yield, ?Await}] \text{ OptionalChain}[?\text{Yield, ?Await}]
  \text{CallExpression}[?\text{Yield, ?Await}] \text{ OptionalChain}[?\text{Yield, ?Await}]
  \text{OptionalExpression}[?\text{Yield, ?Await}] \text{ OptionalChain}[?\text{Yield, ?Await}]

\[\text{OptionalChain}[\text{Yield, Await}] :
  ?. \text{Arguments}[?\text{Yield, ?Await}]
  ?. \text{[ Expression }[+\text{In, ?Yield, ?Await}] \text{ ]}
  ?. \text{IdentifierName}
  ?. \text{TemplateLiteral}[?\text{Yield, ?Await, +Tagged}]
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{ Arguments}[?\text{Yield, ?Await}]
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{ [ Expression }[+\text{In, ?Yield, ?Await}] \text{ ]}
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{ . IdentifierName}
  \text{OptionalChain}[?\text{Yield, ?Await}] \text{ TemplateLiteral}[?\text{Yield, ?Await, +Tagged}]

\[\text{LeftHandSideExpression}[\text{Yield, Await}] :
  \text{NewExpression}[?\text{Yield, ?Await}]
  \text{CallExpression}[?\text{Yield, ?Await}]
  \text{OptionalExpression}[?\text{Yield, ?Await}]

\[\text{UpdateExpression}[\text{Yield, Await}] :
  \text{LeftHandSideExpression}[?\text{Yield, ?Await}]
  \text{LeftHandSideExpression}[?\text{Yield, ?Await}] \text{ [no LineTerminator here]} ++\]
LeftHandSideExpression

++ UnaryExpression
-- UnaryExpression

UnaryExpression : UpdateExpression
delete UnaryExpression
void UnaryExpression
typeof UnaryExpression
+
- ~
! [+Await] AwaitExpression

ExponentiationExpression : UnaryExpression
UpdateExpression ** ExponentiationExpression

MultiplicativeExpression : ExponentiationExpression
MultiplicativeOperator ExponentiationExpression

MultiplicativeOperator : one of
* / %

AdditiveExpression : MultiplicativeExpression
AdditiveExpression + MultiplicativeExpression
AdditiveExpression - MultiplicativeExpression

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

RelationalExpression : ShiftExpression
RelationalExpression < ShiftExpression
RelationalExpression > ShiftExpression
RelationalExpression <= ShiftExpression
RelationalExpression >= ShiftExpression
RelationalExpression instanceof ShiftExpression
[+In] RelationalExpression in ShiftExpression

EqualityExpression : RelationalExpression

EqualityExpression [In, Yield, Await]

BitwiseANDExpression[In, Yield, Await] :
  EqualityExpression[?In, ?Yield, ?Await]

BitwiseXORExpression[In, Yield, Await] :
  BitwiseANDExpression[?In, ?Yield, ?Await]

BitwiseORExpression[In, Yield, Await] :
  BitwiseXORExpression[?In, ?Yield, ?Await]

LogicalANDExpression[In, Yield, Await] :
  BitwiseORExpression[?In, ?Yield, ?Await]
  LogicalANDExpression[?In, ?Yield, ?Await]

LogicalORExpression[In, Yield, Await] :
  LogicalANDExpression[?In, ?Yield, ?Await]
  LogicalANDExpression[?In, ?Yield, ?Await]
  LogicalORExpression[?In, ?Yield, ?Await] || LogicalANDExpression[?In, ?Yield, ?Await]

CoalesceExpression[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]
  AssignmentExpression[?In, ?Yield, ?Await]

AssignmentExpression[In, Yield, Await] :
  ConditionalExpression[?In, ?Yield, ?Await]
  [+Yield] YieldExpression[?In, ?Await]
  ArrowFunction[?In, ?Yield, ?Await]
  AsyncArrowFunction[?In, ?Yield, ?Await]
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 :
  AssignmentExpression

Expression , AssignmentExpression

A.3 Statements
Statement :
  BlockStatement
  VariableStatement
  EmptyStatement
  ExpressionStatement
  IfStatement
  BreakableStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  WithStatement
  LabelledStatement
  TryStatement
  DebuggerStatement

Declaration :
  HoistableDeclaration
  ClassDeclaration
  LexicalDeclaration

HoistableDeclaration :
  FunctionDeclaration
  GeneratorDeclaration
  AsyncFunctionDeclaration
  AsyncGeneratorDeclaration

BreakableStatement :
  IterationStatement
  SwitchStatement

BlockStatement :
  Block

Block :
  { StatementList opt }

StatementList :
  StatementList
  StatementListItem

StatementListItem :
  StatementList
  StatementListItem
Statement[?Yield, ?Await, ?Return]
Declaration[?Yield, ?Await]
LexicalDeclaration[In, Yield, Await]

LetOrConst BindingList[?In, ?Yield, ?Await] ;

LetOrConst :
  let
c

const

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

LexicalBinding[In, Yield, Await]

VariableStatement[Yield, Await]
  var VariableDeclarationList[+In, ?Yield, ?Await] ;

VariableDeclarationList[In, Yield, Await]
  VariableDeclaration[?In, ?Yield, ?Await]

VariableDeclaration[In, Yield, Await]

BindingPattern[Yield, Await]
  ObjectBindingPattern[?Yield, ?Await]
  ArrayBindingPattern[?Yield, ?Await]

ObjectBindingPattern[Yield, Await]
  {
    BindingRestProperty[?Yield, ?Await] 
    BindingPropertyList[?Yield, ?Await] 
  }

ArrayBindingPattern[Yield, Await]
  [ Elision opt BindingRestElement[?Yield, ?Await] opt ]

BindingRestProperty[Yield, Await]
  ... BindingIdentifier[?Yield, ?Await]

BindingPropertyList[Yield, Await]
  BindingProperty[?Yield, ?Await]

BindingElementList[Yield, Await]
  BindingElisionElement[?Yield, ?Await]
BindingElisionElement[Yield, Await] :
   Elision opt BindingElement[Yield, Await]

BindingProperty[Yield, Await] :
   SimpleNameBinding[Yield, Await]
   PropertyName[Yield, Await] : BindingElement[Yield, Await]

BindingElement[Yield, Await] :
   SimpleNameBinding[Yield, Await]
   BindingPattern[Yield, Await] Initializer[+In, Yield, Await] opt

SimpleNameBinding[Yield, Await] :
   BindingIdentifier[Yield, Await] Initializer[+In, Yield, Await] opt

BindingRestElement[Yield, Await] Initializer[+In, Yield, Await] opt
   ... 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 ( [lookahead ∉ { let, async of }] LeftHandSideExpression[^Yield, ?Await] of
for ( ForDeclaration[^Yield, ?Await] of AssignmentExpression[^+In, ?Yield, ?Await] )
  [+Await] for await ( [lookahead ≠ let] LeftHandSideExpression[^Yield, ?Await] of
ForDeclaration[^Yield, Await] :
  LetOrConst ForBinding[^Yield, ?Await]
ForBinding[^Yield, Await] :
  BindingIdentifier[^Yield, ?Await]
  BindingPattern[^Yield, ?Await]
ContinueStatement[^Yield, Await] :
  continue ;
BreakStatement[^Yield, Await] :
  break ;
ReturnStatement[^Yield, Await] :
  return ;
WithStatement[^Yield, Await, Return] :
SwitchStatement[^Yield, Await, Return] :
CaseBlock[^Yield, Await, Return] :
  { CaseClauses[^Yield, ?Await, ?Return] opt }
CaseClauses[^Yield, Await, Return] :
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] :
```javascript
function BindingIdentifier (?Yield, ?Await) ( FormalParameters [~Yield, ~Await] ) {
    FunctionBody [~Yield, ~Await]
}

[+Default] function ( FormalParameters [~Yield, ~Await] ) { FunctionBody [~Yield, ~Await] }

FunctionDeclaration [Yield, Await, Default] :
    function BindingIdentifier (?Yield, ?Await) ( 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


the interpretation of CoverParenthesizedExpressionAndArrowParameterList is refined using the following grammar:

ArrowFormalParameters [Yield, Await] :
    ( UniqueFormalParameters [Yield, ?Await] )

AsyncArrowFunction [In, Yield, Await] :
    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] :
```

*832*
When processing an instance of the production
\( \text{AsyncArrowFunction} [\text{In, Yield, Await}] : \text{CoverCallExpressionAndAsyncArrowHead} [?\text{Yield, ?Await}] \quad \text{[no LineTerminator here]} \Rightarrow \text{AsyncConciseBody} [?\text{In}] \)

the interpretation of \( \text{CoverCallExpressionAndAsyncArrowHead} \) is refined using the following grammar:

\[
\text{AsyncArrowHead} :
\]

\[
\text{MethodDefinition} [\text{Yield, Await}] :
    \text{PropertyName} [?\text{Yield, ?Await}] ( \text{UniqueFormalParameters} [~\text{Yield, ~Await}] ) \{ \text{FunctionBody} [~\text{Yield, ~Await}] \}
\]

\[
\text{GeneratorMethod} [?\text{Yield, ?Await}]
\]

\[
\text{AsyncMethod} [?\text{Yield, ?Await}]
\]

\[
\text{AsyncGeneratorMethod} [?\text{Yield, ?Await}]
\]

\[
\text{get PropertyName} [?\text{Yield, ?Await}] ( ) \{ \text{FunctionBody} [~\text{Yield, ~Await}] \}
\]

\[
\text{set PropertyName} [?\text{Yield, ?Await}] ( \text{PropertySetParameterList} ) \{ \text{FunctionBody} [~\text{Yield, ~Await}] \}
\]

\[
\text{PropertySetParameterList} :
    \text{FormalParameter [~Yield, ~Await]}
\]

\[
\text{GeneratorMethod} [?\text{Yield, Await}] :
    \ast \text{PropertyName} [?\text{Yield, ?Await}] ( \text{UniqueFormalParameters} [+\text{Yield, ~Await}] ) \{ \text{GeneratorBody} \}
\]

\[
\text{GeneratorDeclaration} [?\text{Yield, Await, Default}] :
    \text{function} \ast \text{BindingIdentifier} [?\text{Yield, ?Await}] ( \text{FormalParameters} [+\text{Yield, ~Await}] ) \{ \text{GeneratorBody} \}
\]

\[
\text{ [+Default]} \text{function} \ast ( \text{FormalParameters} [+\text{Yield, ~Await}] ) \{ \text{GeneratorBody} \}
\]

\[
\text{GeneratorExpression} :
    \text{function} \ast \text{BindingIdentifier} [+\text{Yield, ~Await}] \text{opt ( FormalParameters} [+\text{Yield, ~Await}] ) \{ \text{GeneratorBody} \}
\]

\[
\text{GeneratorBody} :
    \text{FunctionBody [+Yield, ~Await]}
\]

\[
\text{YieldExpression} [\text{In, Await}] :
    \text{yield}
\]

\[
\text{yield [no LineTerminator here] AssignmentExpression} [?\text{In, +Yield, ?Await}]
\]

\[
\text{yield [no LineTerminator here] \ast AssignmentExpression} [?\text{In, +Yield, ?Await}]
\]

\[
\text{AsyncGeneratorMethod} [\text{Yield, Await}] :
    \text{async [no LineTerminator here] \ast PropertyName} [?\text{Yield, ?Await}] ( \text{UniqueFormalParameters} [+\text{Yield, +Await}] ) \{ \text{AsyncGeneratorBody} \}
\]

\[
\text{AsyncGeneratorDeclaration} [\text{Yield, Await, Default}] :
    \text{async [no LineTerminator here] function} \ast \text{BindingIdentifier} [?\text{Yield, ?Await}] ( \text{FormalParameters} [+\text{Yield, +Await}] ) \{ \text{AsyncGeneratorBody} \}
\]

\[
\text{ [+Default]} \text{async [no LineTerminator here] function} \ast ( \text{FormalParameters} [+\text{Yield, +Await}] ) \{ \text{AsyncGeneratorBody} \}
\]

\[
\text{AsyncGeneratorExpression} :
    \text{async [no LineTerminator here] function} \ast \text{BindingIdentifier} [+\text{Yield, +Await}] \text{opt ( FormalParameters} [+\text{Yield, +Await}] ) \{ \text{AsyncGeneratorBody} \}
\]

\[
\text{AsyncGeneratorBody} :
\]

833
A.5 Scripts and Modules

Script:
  ScriptBody opt

ScriptBody:
  StatementList (~Yield, ~Await, ~Return)

Module:
ModuleBody
ModuleBody : ModuleItemList
ModuleItemList : ModuleItem ModuleItemList 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 :::
  StrWhiteSpace opt
  StrWhiteSpace opt StrNumericLiteral StrWhiteSpace opt

StrWhiteSpace :::
  StrWhiteSpaceChar StrWhiteSpace opt

StrWhiteSpaceChar :::
  WhiteSpace
  LineTerminator

StrNumericLiteral :::
  StrDecimalLiteral
  NonDecimalIntegerLiteral [~Sep]

StrDecimalLiteral :::
  StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :::
  Infinity
  . 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

\[
\begin{align*}
\text{uri} & \ ::= \ 
\text{uriCharacters}_{\text{opt}} \\
\text{uriCharacters} & \ ::= \ 
\text{uriCharacter} \ \text{uriCharacters}_{\text{opt}} \\
\text{uriCharacter} & \ ::= \ 
\text{uriReserved} \\
& \quad \text{uriUnescaped} \\
& \quad \text{uriEscaped} \\
\text{uriReserved} & \ ::= \ \textbf{one of} \\
& \quad ; / ? : @ & = + $ , \\
\text{uriUnescaped} & \ ::= \ 
\text{uriAlpha} \\
& \quad \text{DecimalDigit} \\
& \quad \text{uriMark} \\
\text{uriEscaped} & \ ::= \ 
\% \text{HexDigit} \text{ HexDigit} \\
\text{uriAlpha} & \ ::= \ \textbf{one of} \\
& \quad a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
\text{uriMark} & \ ::= \ \textbf{one of} \\
& \quad - _ . ! ~ * ' ( 
\end{align*}
\]

A.8 Regular Expressions

\[
\begin{align*}
\text{Pattern}_{[U, \ N]} & \ ::= \ 
\text{Disjunction}_{[?U, \ ?N]} \\
\text{Disjunction}_{[U, \ N]} & \ ::= \ 
\text{Alternative}_{[?U, \ ?N]} \\
& \quad \text{Alternative}_{[?U, \ ?N]} \ | \ \text{Disjunction}_{[?U, \ ?N]} \\
\text{Alternative}_{[U, \ N]} & \ ::= \ 
[\text{empty}] \\
& \quad \text{Alternative}_{[?U, \ ?N]} \ \text{Term}_{[?U, \ ?N]} \\
\text{Term}_{[U, \ N]} & \ ::= \ 
\text{Assertion}_{[?U, \ ?N]} \\
& \quad \text{Atom}_{[?U, \ ?N]} \\
& \quad \text{Atom}_{[?U, \ ?N]} \ \text{Quantifier} \\
\text{Assertion}_{[U, \ N]} & \ ::= \ 
^ \\
& \quad $ \\
& \quad \backslash b \\
& \quad \backslash B \\
& \quad ( ? = \text{Disjunction}_{[?U, \ ?N]} ) \\
& \quad ( ? ! \text{Disjunction}_{[?U, \ ?N]} ) 
\end{align*}
\]
( ? <= 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 \[?u\]

838
GroupName \[
< \text{RegExpIdentifierName} \] >

RegExpIdentifierName \[
\text{RegExpIdentifierStart} \quad \text{RegExpIdentifierName} \ 	ext{RegExpIdentifierPart}
\]

RegExpIdentifierPart \[
\text{UnicodeIDStart} \\
\text{UnicodeIDContinue}
\]

RegExpUnicodeEscapeSequence \[
\text{UnicodeLeadSurrogate} \ 	ext{UnicodeTrailSurrogate}
\]

UnicodeLeadSurrogate \[
\text{any Unicode code point in the inclusive range 0xD800 to 0xDBFF}
\]

UnicodeTrailSurrogate \[
\text{any Unicode code point in the inclusive range 0xDC00 to 0xDFFF}
\]

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.

\text{HexLeadSurrogate} ::

\text{Hex4Digits} but only if the MV of \text{Hex4Digits} is in the inclusive range 0xD800 to 0xDBFF

\text{HexTrailSurrogate} ::

\text{Hex4Digits} but only if the MV of \text{Hex4Digits} is in the inclusive range 0xDC00 to 0xDFFF

\text{HexNonSurrogate} ::

\text{Hex4Digits} but only if the MV of \text{Hex4Digits} is not in the inclusive range 0xD800 to 0xDFFF

\text{IdentityEscape} \[
\text{SyntaxCharacter}
\]

\text{SourceCharacter} but not \text{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]
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

\[
\text{NumericLiteral} :: \ \\
\text{DecimalLiteral} \\
\text{DecimalBigIntegerLiteral}
\]
NonDecimalIntegerLiteral
NonDecimalIntegerLiteral 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 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 \not\in OctalDigit]
  ZeroToThree OctalDigit [lookahead \not\in 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:

```javascript
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).
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

```plaintext
Comment ::
    MultiLineComment
    SingleLineComment
    SingleLineHTMLOpenComment
    SingleLineHTMLCloseComment
    SingleLineDelimitedComment

MultiLineComment ::
    /* FirstCommentLine opt \n        LineTerminator MultiLineCommentChars opt */ \n        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 ::
```
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] 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] 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] )
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 \(\backslash\) or \(\) or \(-\)
\(\backslash\) ClassEscape \([?U, ?N]\)
\([-\text{lookahead} = c]\)

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:

\[\text{ClassAtomNoDash} :: \text{\textbackslash} \ \left[ \text{lookahead} = \text{c} \right]\]

1. Return false.

B.1.4.3 Static Semantics: CharacterValue

The semantics of 22.2.1.4 is extended as follows:

\[\text{ClassAtomNoDash} :: \text{\textbackslash} \ \left[ \text{lookahead} = \text{c} \right]\]

1. Return the code point value of U+005C (REVERSE SOLIDUS).

\[\text{ClassEscape} :: \text{c} \ \text{ClassControlLetter}\]

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.

\[\text{CharacterEscape} :: \text{LegacyOctalEscapeSequence}\]

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 \(\text{Term} :: \text{QuantifiableAssertion Quantifier}\) evaluates the same as the production \(\text{Term} :: \text{Atom Quantifier}\) but with QuantifiableAssertion substituted for Atom.

The production \(\text{Term} :: \text{ExtendedAtom Quantifier}\) evaluates the same as the production \(\text{Term} :: \text{Atom Quantifier}\) but with ExtendedAtom substituted for Atom.

The production \(\text{Term} :: \text{ExtendedAtom}\) evaluates the same as the production \(\text{Term} :: \text{Atom}\) but with ExtendedAtom substituted for Atom.

Assertion (22.2.2.6) includes the following additional evaluation rule:

The production \(\text{Assertion} :: \text{QuantifiableAssertion}\) evaluates as follows:

1. Evaluate \(\text{QuantifiableAssertion}\) to obtain a Matcher \(m\).
2. Return \(m\).

Assertion (22.2.2.6) evaluation rules for the \(\text{Assertion} :: ( \ ? = \text{Disjunction} )\) and \(\text{Assertion} :: ( \ ? ! \text{Disjunction} )\) productions are also used for the \(\text{QuantifiableAssertion}\) 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 ! CharSetMatcher(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.

CharacterEscape (22.2.10) 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 (22.2.15) 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 (22.2.16) 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 (22.2.19) 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 (22.2.18) 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 ( \textit{A}, \textit{B} )

The abstract operation CharacterRangeOrUnion takes arguments \textit{A} (a CharSet) and \textit{B} (a CharSet). It performs the following steps when called:

1. If Unicode is false, then
   a. If \textit{A} does not contain exactly one character or \textit{B} does not contain exactly one character, then
      i. Let \textit{C} be the CharSet containing the single character \textbackslash U+002D (HYPHEN-MINUS).
      ii. Return the union of CharSets \textit{A}, \textit{B} and \textit{C}.
2. Return ! CharacterRange(\textit{A}, \textit{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

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Global Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%escape%</td>
<td>escape</td>
<td>The escape function (B.2.1.1)</td>
</tr>
<tr>
<td>%unescape%</td>
<td>unescape</td>
<td>The unescape function (B.2.1.2)</td>
</tr>
</tbody>
</table>

B.2.1.1 escape ( \textit{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 \texttt{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 \texttt{0x00FF}, a four-digit escape sequence of the form \%uuxxxx is used.

The escape function is the \%escape\% intrinsic object. When the escape function is called with one argument \textit{string}, the following steps are taken:

1. Set \textit{string} to ? ToString(\textit{string}).
2. Let length be the number of code units in \textit{string}.
3. Let \textit{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 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./", then
      i. Let $S$ be the String value containing the single code unit char.
   c. Else if char ≥ 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: 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 \( \text{skip} \) to 5.

iv. Else if \( k \leq \text{length} - 3 \), then
   1. Set \( \text{hexEscape} \) to the substring of \( \text{string} \) from \( k + 1 \) to \( k + 3 \).
   2. Set \( \text{skip} \) to 2.

v. If \( \text{hexEscape} \) can be interpreted as an expansion of \( \text{HexDigits} \), then
   1. Let \( \text{hexIntegerLiteral} \) be the string-concatenation of \( "0x" \) and \( \text{hexEscape} \).
   2. Let \( n \) be \! \( \text{ToNumber}(\text{hexIntegerLiteral}) \).
   3. Set \( c \) to the code unit whose value is \( R(n) \).
   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 \? \( \text{ToObject}(\text{this} \text{ value}) \).
2. Return \( O.\text{[[GetPrototypeOf]]}() \).

**B.2.2.1.2 set Object.prototype.__proto__**

The value of the [[Set]] attribute is a built-in function that takes an argument \( \text{proto} \). It performs the following steps when called:

1. Let \( O \) be \? \( \text{RequireObjectCoercible}(\text{this} \text{ value}) \).
2. If \( \text{Type}(\text{proto}) \) is neither Object nor Null, return \( \text{undefined} \).
3. If \( \text{Type}(O) \) is not Object, return \( \text{undefined} \).
4. Let \( \text{status} \) be \? \( O.\text{[[SetPrototypeOf]]}(\text{proto}) \).
5. If \( \text{status} \) is \( \text{false} \), throw a \( \text{TypeError} \) exception.
6. Return \( \text{undefined} \).

#### B.2.2.2 Object.prototype.__defineGetter__ ( \( P, \text{getter} \) )

When the \( \text{__defineGetter__} \) method is called with arguments \( P \) and \( \text{getter} \), the following steps are taken:

1. Let \( O \) be \? \( \text{ToObject}(\text{this} \text{ value}) \).
2. If \( \text{IsCallable}(\text{getter}) \) is \( \text{false} \), throw a \( \text{TypeError} \) exception.
3. Let \( \text{desc} \) be PropertyDescriptor { [[Get]]: \( \text{getter} \), [[Enumerable]]: true, [[Configurable]]: true }.
4. Let \( \text{key} \) be \? \( \text{ToPropertyKey}(P) \).
5. Perform \( \text{DefinePropertyOrThrow}(O, \text{key}, \text{desc}) \).
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 $\text{substr}$ 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 "&quot;.
   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 ">".
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.

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.

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.

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.

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.

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; 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 ≤ \( yi \) ≤ 99, let yyyy be 1900 + 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.
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 $\text{RequireInternalSlot}(O, \text{[[RegExpMatcher]]})$.
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.\text{[[OriginalSource]]}.
   c. Let $F$ be pattern.\text{[[OriginalFlags]]}.
4. Else,
   a. Let $P$ be pattern.
   b. Let $F$ be flags.
5. Return $\text{RegExpInitialize}(O, P, F)$.

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.

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 :

\[
\{ \text{PropertyDefinitionList} \} \\
\{ \text{PropertyDefinitionList} , \}
\]

- 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.

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 `nullNamedEvaluation` 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 `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 \( g \)
- 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 `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 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 \( \text{func} \) 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 \( \text{initializedBindings} \) does not contain \( F \) and \( F \) is not "arguments", then
   a. Perform \( \text{varEnv}.\text{CreateMutableBinding}(F, \text{false}) \).
   b. Perform \( \text{varEnv}.\text{InitializeBinding}(F, \text{undefined}) \).
   c. Append \( F \) to \( \text{instantiatedVarNames} \).

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 \( ! \text{benv}.\text{GetBindingValue}(F, \text{false}) \).
   d. Perform \( ! \text{fenv}.\text{SetMutableBinding}(F, fobj, \text{false}) \).
   e. Return NormalCompletion(\( \text{empty} \)).

B.3.3.2 Changes to GlobalDeclarationInstantiation

During GlobalDeclarationInstantiation the following steps are performed in place of step 13:

13. Let \( \text{strict} \) be IsStrict of \( \text{script} \).

14. If \( \text{strict} \) is false, then
   a. Let \( \text{declaredFunctionOrVarNames} \) be a new empty List.
   b. Append to \( \text{declaredFunctionOrVarNames} \) the elements of \( \text{declaredFunctionNames} \).
   c. Append to \( \text{declaredFunctionOrVarNames} \) the elements of \( \text{declaredVarNames} \).
   d. For each FunctionDeclaration \( f \) that is directly contained in the StatementList of a Block, CaseClause, or DefaultClause Contained within \( \text{script} \), 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 \( \text{script} \), then
         1. If \( \text{env}.\text{HasLexicalDeclaration}(F) \) is false, then
            a. Let \( \text{fnDefinable} \) be \( ? \text{env}.\text{CanDeclareGlobalVar}(F) \).
            b. If \( \text{fnDefinable} \) 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 \( \text{declaredFunctionOrVarNames} \) does not contain \( F \), then
                  i. Perform \( ? \text{env}.\text{CreateGlobalVarBinding}(F, \text{false}) \).
                  ii. Append \( F \) to \( \text{declaredFunctionOrVarNames} \).
               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 \( ! \text{benv}.\text{GetBindingValue}(F, \text{false}) \).
                  iv. Perform \( ? \text{genv}.\text{SetMutableBinding}(F, fobj, \text{false}) \).
                  v. Return NormalCompletion(\( \text{empty} \)).

B.3.3.3 Changes to EvalDeclarationInstantiation

During EvalDeclarationInstantiation the following steps are performed in place of step 7:

7. If \( \text{strict} \) 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 `false`.
         b. Else,
            i. Let `fnDefinable` be `true`.
      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`).
            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

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
iv. Else,
   1. Assert: d is a FunctionDeclaration.

B.3.4 FunctionDeclarations in IfStatement Statement Clauses

The following augments the IfStatement production in 14.6:

IfStatement[Yield, Await, Return] :

Statement[?Yield, ?Await, ?Return]
FunctionDeclaration[?Yield, ?Await, ~Default]
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.
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:

1. 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:

1. 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


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).
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.
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

<table>
<thead>
<tr>
<th>Type of val</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object (has an [[IsHTMLDDA]] internal slot)</td>
<td>&quot;undefined&quot;</td>
</tr>
</tbody>
</table>

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 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(...)

867
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

9.1.4.15-9.1.1.4.18 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 \+0F or -0F as the representation of a 0 time value. ECMAScript 2015 specifies that \+0F always returned. This means that for ECMAScript 2015 the time value of a Date object is never observably -0F and methods that return time values never return -0F.

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 as value is treated as if \+0F 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
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.getOwnPropertyDescriptor` is not an object an attempt is made to coerce the argument using `ToObject`. If the coercion is successful the result is used in place of the original argument value. In 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.getOwnPropertyNames` is not an object an attempt is made to coerce the argument using `ToObject`. If the coercion is successful the result is used in place of the original argument value. In 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 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.

20.1.2.14: 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 exists 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 Ecmarkup. Ecmarkup 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. Ecmarkup builds on and integrates a number of other formats and technologies including Grammarkdown for defining syntax and Ecmarkdown 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 Ecmarkup source text that formed the basis of this edition was produced by converting the ECMAScript 2015 Word document to Ecmarkup using an automated conversion tool.

H Bibliography


   NOTE There are no normative changes between IEEE 754-2008 and IEEE 754-2019 that affect the ECMA-262 specification.

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

I Copyright & Software License

Ecma International
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.