 COPYRIGHT NOTICE

© 2017 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 http://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.
Introduction
1 Scope
2 Conformance
3 Normative References
4 Overview
  4.1 Web Scripting
  4.2 ECMAScript Overview
    4.2.1 Objects
    4.2.2 The Strict Variant of ECMAScript
4.3 Terms and Definitions
  4.3.1 type
  4.3.2 primitive value
  4.3.3 object
  4.3.4 constructor
  4.3.5 prototype
  4.3.6 ordinary object
  4.3.7 exotic object
  4.3.8 standard object
  4.3.9 built-in object
  4.3.10 undefined value
  4.3.11 Undefined type
  4.3.12 null value
  4.3.13 Null type
  4.3.14 Boolean value
  4.3.15 Boolean type
  4.3.16 Boolean object
  4.3.17 String value
  4.3.18 String type
  4.3.19 String object
  4.3.20 Number value
  4.3.21 Number type
  4.3.22 Number object
  4.3.23 Infinity
4.3.24 NaN
4.3.25 Symbol value
4.3.26 Symbol type
4.3.27 Symbol object
4.3.28 function
4.3.29 built-in function
4.3.30 property
4.3.31 method
4.3.32 built-in method
4.3.33 attribute
4.3.34 own property
4.3.35 inherited property

4.4 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

5.3 Static Semantic Rules

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
6.1.5 The Symbol Type
  6.1.5.1 Well-Known Symbols
6.1.6 The Number Type
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 NormalCompletion
6.2.3.2 Implicit Completion Values
6.2.3.3 Throw an Exception
6.2.3.4 ReturnIfAbrupt
6.2.3.5 UpdateEmpty ( completionRecord, value )

6.2.4 The Reference Specification Type
   6.2.4.1 GetValue ( V )
   6.2.4.2 PutValue ( V, W )
   6.2.4.3 GetThisValue ( V )
   6.2.4.4 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 Lexical Environment and Environment Record Specification Types

6.2.7 Data Blocks
   6.2.7.1 CreateByteDataBlock ( size )
   6.2.7.2 CreateSharedByteDataBlock ( size )
   6.2.7.3 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 ToNumber ( argument )
      7.1.3.1 ToNumber Applied to the String Type
      7.1.3.1.1 RS: MV
   7.1.4 ToInteger ( argument )
   7.1.5 ToInt32 ( argument )
   7.1.6 ToUint32 ( argument )
   7.1.7 ToInt16 ( argument )
   7.1.8 ToUint16 ( argument )
   7.1.9 ToInt8 ( argument )
   7.1.10 ToUint8 ( argument )
   7.1.11 To Uint8Clamp ( argument )
   7.1.12 ToString ( argument )
      7.1.12.1 ToString Applied to the Number Type
7.1.13 ToObject (argument)
7.1.14 ToPropertyKey (argument)
7.1.15 ToLength (argument)
7.1.16 CanonicalNumericIndexString (argument)
7.1.17 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 IsInteger (argument)
7.2.7 IsPropertyKey (argument)
7.2.8 IsRegExp (argument)
7.2.9 SameValue (x, y)
7.2.10 SameValueZero (x, y)
7.2.11 SameValueNonNumber (x, y)
7.2.12 Abstract Relational Comparison
7.2.13 Abstract Equality Comparison
7.2.14 Strict Equality Comparison

7.3 Operations on Objects
7.3.1 Get (O, P)
7.3.2 GetV (V, P)
7.3.3 Set (O, P, V, Throw)
7.3.4 CreateDataProperty (O, P, V)
7.3.5 CreateMethodProperty (O, P, V)
7.3.6 CreateDataPropertyOrThrow (O, P, V)
7.3.7 DefinePropertyOrThrow (O, P, desc)
7.3.8 DeletePropertyOrThrow (O, P)
7.3.9 GetMethod (V, P)
7.3.10 HasProperty (O, P)
7.3.11 HasOwnProperty (O, P)
7.3.12 Call (F, V[ , argumentsList ])
7.3.13 Construct (F[ , argumentsList [ , newTarget ]])
7.3.14 SetIntegrityLevel (O, level)
7.3.15 TestIntegrityLevel (O, level)
7.3.16 CreateArrayFromList (elements)
7.3.17 CreateListFromArrayLike (obj [ , elementTypes ])
7.3.18 Invoke (V, P[ , argumentsList ])
7.3.19 OrdinaryHasInstance (C, O)
7.3.20 SpeciesConstructor ( O, defaultConstructor )
7.3.21 EnumerableOwnProperties ( O, kind )
7.3.22 GetFunctionRealm ( obj )

7.4 Operations on Iterator Objects
7.4.1 GetIterator ( obj [ , method ] )
7.4.2 IteratorNext ( iterator [ , value ] )
7.4.3 IteratorComplete ( iterResult )
7.4.4 IteratorValue ( iterResult )
7.4.5 IteratorStep ( iterator )
7.4.6 IteratorClose ( iterator, completion )
7.4.7 CreateIterResultObject ( value, done )
7.4.8 CreateListIterator ( list )
7.4.8.1 ListIterator next()

8 Executable Code and Execution Contexts
8.1 Lexical Environments
8.1.1 Environment Records
8.1.1.1 Declarative Environment Records
8.1.1.1.1 HasBinding ( N )
8.1.1.1.2 CreateMutableBinding ( N, D )
8.1.1.1.3 CreateImmutableBinding ( N, S )
8.1.1.1.4 InitializeBinding ( N, V )
8.1.1.1.5 SetMutableBinding ( N, V, S )
8.1.1.1.6 GetBindingValue ( N, S )
8.1.1.1.7 DeleteBinding ( N )
8.1.1.1.8 HasThisBinding ( )
8.1.1.1.9 HasSuperBinding ( )
8.1.1.1.10 WithBaseObject ( )

8.1.1.2 Object Environment Records
8.1.1.2.1 HasBinding ( N )
8.1.1.2.2 CreateMutableBinding ( N, D )
8.1.1.2.3 CreateImmutableBinding ( N, S )
8.1.1.2.4 InitializeBinding ( N, V )
8.1.1.2.5 SetMutableBinding ( N, V, S )
8.1.1.2.6 GetBindingValue ( N, S )
8.1.1.2.7 DeleteBinding ( N )
8.1.1.2.8 HasThisBinding ( )
8.1.1.2.9 HasSuperBinding ( )
8.1.1.2.10 WithBaseObject ( )

8.1.1.3 Function Environment Records
8.1.1.3.1 BindThisValue ( V )
8.1.1.3.2 HasThisBinding ( )
8.1.1.3.3 HasSuperBinding ( )
8.1.1.3.4 GetThisBinding ( )
8.1.1.3.5 GetSuperBase ( )

8.1.1.4 Global Environment Records
8.1.1.4.1 HasBinding ( N )
8.1.1.4.2 CreateMutableBinding ( N, D )
8.1.1.4.3 CreateImmutableBinding ( N, S )
8.1.1.4.4 InitializeBinding ( N, V )
8.1.1.4.5 SetMutableBinding ( N, V, S )
8.1.1.4.6 GetBindingValue ( N, S )
8.1.1.4.7 DeleteBinding ( N )
8.1.1.4.8 HasThisBinding ( )
8.1.1.4.9 HasSuperBinding ( )
8.1.1.4.10 WithBaseObject ( )
8.1.1.4.11 GetThisBinding ( )
8.1.1.4.12 HasVarDeclaration ( N )
8.1.1.4.13 HasLexicalDeclaration ( N )
8.1.1.4.14 HasRestrictedGlobalProperty ( N )
8.1.1.4.15 CanDeclareGlobalVar ( N )
8.1.1.4.16 CanDeclareGlobalFunction ( N )
8.1.1.4.17 CreateGlobalVarBinding ( N, D )
8.1.1.4.18 CreateGlobalFunctionBinding ( N, V, D )

8.1.1.5 Module Environment Records
8.1.1.5.1 GetBindingValue ( N, S )
8.1.1.5.2 DeleteBinding ( N )
8.1.1.5.3 HasThisBinding ( )
8.1.1.5.4 GetThisBinding ( )
8.1.1.5.5 CreateImportBinding ( N, M, N2 )

8.1.2 Lexical Environment Operations
8.1.2.1 GetIdentifierReference ( lex, name, strict )
8.1.2.2 NewDeclarativeEnvironment ( E )
8.1.2.3 NewObjectEnvironment ( O, E )
8.1.2.4 NewFunctionEnvironment ( F, newTarget )
8.1.2.5 NewGlobalEnvironment ( G, thisValue )
8.1.2.6 NewModuleEnvironment ( E )

8.2 Realms
8.2.1 CreateRealm ( )
8.2.2 CreateIntrinsics ( realmRec )
8.2.3 SetRealmGlobalObject ( realmRec, globalObj, thisValue )
8.2.4 SetDefaultGlobalBindings (realmRec)

8.3 Execution Contexts
8.3.1 GetActiveScriptOrModule ()
8.3.2 ResolveBinding (name, env)
8.3.3 GetThisEnvironment ()
8.3.4 ResolveThisBinding ()
8.3.5 GetNewTarget ()
8.3.6 GetGlobalObject ()

8.4 Jobs and Job Queues
8.4.1 EnqueueJob (queueName, job, arguments)

8.5 InitializeHostDefinedRealm ()
8.6 RunJobs ()

8.7 Agents
8.7.1 AgentSignifier ()
8.7.2 AgentCanSuspend ()

8.8 Agent Clusters

8.9 Forward Progress

9 Ordinary and Exotic Objects Behaviours
9.1 Ordinary Object Internal Methods and Internal Slots
9.1.1 [[GetPrototypeOf]] ()
"9.1.1.1 OrdinaryGetPrototypeOf (O)
9.1.2 [[SetPrototypeOf]] (V)
"9.1.2.1 OrdinarySetPrototypeOf (O, V)
9.1.3 [[IsExtensible]] ()
"9.1.3.1 OrdinaryIsExtensible (O)
9.1.4 [[PreventExtensions]] ()
"9.1.4.1 OrdinaryPreventExtensions (O)
9.1.5 [[GetOwnProperty]] (P)
"9.1.5.1 OrdinaryGetOwnProperty (O, P)
9.1.6 [[DefineOwnProperty]] (P, Desc)
"9.1.6.1 OrdinaryDefineOwnProperty (O, P, Desc)
9.1.6.2 IsCompatiblePropertyDescriptor (Extensible, Desc, Current)
"9.1.6.3 ValidateAndApplyPropertyDescriptor (O, P, extensible, Desc, current)
9.1.7 [[HasProperty]] (P)
"9.1.7.1 OrdinaryHasProperty (O, P)
9.1.8 [[Get]] (P, Receiver)
"9.1.8.1 OrdinaryGet (O, P, Receiver)
9.1.9 [[Set]] (P, V, Receiver)
"9.1.9.1 OrdinarySet (O, P, V, Receiver)
9.1.10 [[Delete]] (P)
9.1.10.1 OrdinaryDelete (O, P)
9.1.11 [[OwnPropertyKeys]] ()
  9.1.11.1 OrdinaryOwnPropertyKeys (O)
9.1.12 ObjectCreate (proto [, internalSlotsList])
9.1.13 OrdinaryCreateFromConstructor (constructor, intrinsicDefaultProto [, internalSlotsList])
9.1.14 GetPrototypeFromConstructor (constructor, intrinsicDefaultProto)

9.2 ECMAScript Function Objects
  9.2.1 [[Call]] (thisArgument, argumentsList)
    9.2.1.1 PrepareForOrdinaryCall (F, newTarget)
    9.2.1.2 OrdinaryCallBindThis (F, calleeContext, thisArgument)
    9.2.1.3 OrdinaryCallEvaluateBody (F, argumentsList)
  9.2.2 [[Construct]] (argumentsList, newTarget)
  9.2.3 FunctionAllocate (functionPrototype, strict, functionKind)
  9.2.4 FunctionInitialize (F, kind, ParameterList, Body, Scope)
  9.2.5 FunctionCreate (kind, ParameterList, Body, Scope, Strict [, prototype])
  9.2.6 GeneratorFunctionCreate (kind, ParameterList, Body, Scope, Strict)
  9.2.7 AddRestrictedFunctionProperties (F, realm)
    9.2.7.1 %ThrowTypeError% ()
  9.2.8 MakeConstructor (F [, writablePrototype [, prototype]])
  9.2.9 MakeClassConstructor (F)
  9.2.10 MakeMethod (F, homeObject)
  9.2.11 SetFunctionName (F, name [, prefix])
  9.2.12 FunctionDeclarationInstantiation (func, argumentsList)

9.3 Built-in Function Objects
  9.3.1 [[Call]] (thisArgument, argumentsList)
  9.3.2 [[Construct]] (argumentsList, newTarget)
  9.3.3 CreateBuiltInFunction (realm, steps, prototype [, internalSlotsList])

9.4 Built-in Exotic Object Internal Methods and Slots
  9.4.1 Bound Function Exotic Objects
    9.4.1.1 [[Call]] (thisArgument, argumentsList)
    9.4.1.2 [[Construct]] (argumentsList, newTarget)
    9.4.1.3 BoundFunctionCreate (targetFunction, boundThis, boundArgs)
  9.4.2 Array Exotic Objects
    9.4.2.1 [[DefineOwnProperty]] (P, Desc)
    9.4.2.2 ArrayCreate (length [, proto])
    9.4.2.3 ArraySpeciesCreate (originalArray, length)
    9.4.2.4 ArraySetLength (A, Desc)
  9.4.3 String Exotic Objects
    9.4.3.1 [[GetOwnProperty]] (P)
    9.4.3.2 [[DefineOwnProperty]] (P, Desc)
9.4.3.3 `[[OwnPropertyKeys]]()`
9.4.3.4 `StringCreate(value, prototype)`
9.4.3.5 `StringGetOwnProperty(S, P)`

9.4.4 Arguments Exotic Objects
9.4.4.1 `[[GetOwnProperty]](P)`
9.4.4.2 `[[DefineOwnProperty]](P, Desc)`
9.4.4.3 `[[Get]](P, Receiver)`
9.4.4.4 `[[Set]](P, V, Receiver)`
9.4.4.5 `[[Delete]](P)`

9.4.4.6 `CreateUnmappedArgumentsObject(argumentsList)`
9.4.4.7 `CreateMappedArgumentsObject(func, formals, argumentsList, env)`
  9.4.4.7.1 `MakeArgGetter(name, env)`
  9.4.4.7.2 `MakeArgSetter(name, env)`

9.4.5 Integer Indexed Exotic Objects
9.4.5.1 `[[GetOwnProperty]](P)`
9.4.5.2 `[[HasProperty]](P)`
9.4.5.3 `[[DefineOwnProperty]](P, Desc)`
9.4.5.4 `[[Get]](P, Receiver)`
9.4.5.5 `[[Set]](P, V, Receiver)`
9.4.5.6 `[[OwnPropertyKeys]]()`

9.4.5.7 `IntegerIndexedObjectCreate(prototype, internalSlotsList)`
9.4.5.8 `IntegerIndexedElementGet(O, index)`
9.4.5.9 `IntegerIndexedElementSet(O, index, value)`

9.4.6 Module Namespace Exotic Objects
9.4.6.1 `[[SetPrototypeOf]](V)`
9.4.6.2 `[[IsExtensible]]()`
9.4.6.3 `[[PreventExtensions]]()`
9.4.6.4 `[[GetOwnProperty]](P)`
9.4.6.5 `[[DefineOwnProperty]](P, Desc)`
9.4.6.6 `[[HasProperty]](P)`
9.4.6.7 `[[Get]](P, Receiver)`
9.4.6.8 `[[Set]](P, V, Receiver)`
9.4.6.9 `[[Delete]](P)`

9.4.6.10 `[[OwnPropertyKeys]]()`
9.4.6.11 `ModuleNamespaceCreate(module, exports)`

9.4.7 Immutable Prototype Exotic Objects
9.4.7.1 `[[SetPrototypeOf]](V)`
9.4.7.2 `SetImmutablePrototype(O, V)`

9.5 Proxy Object Internal Methods and Internal Slots
9.5.1 `[[GetPrototypeOf]]()`
9.5.2 [[SetPrototypeOf]] (V)
9.5.3 [[IsExtensible]] ()
9.5.4 [[PreventExtensions]] ()
9.5.5 [[GetOwnProperty]] (P)
9.5.6 [[DefineOwnProperty]] (P, Desc)
9.5.7 [[HasProperty]] (P)
9.5.8 [[Get]] (P, Receiver)
9.5.9 [[Set]] (P, V, Receiver)
9.5.10 [[Delete]] (P)
9.5.11 [[OwnPropertyKeys]] ()
9.5.12 [[Call]] (thisArgument, argumentsList)
9.5.13 [[Construct]] (argumentsList, newTarget)
9.5.14 ProxyCreate (target, handler)

10 ECMAScript Language: Source Code

10.1 Source Text
10.1.1 SS: UTF16Encoding (cp)
10.1.2 SS: UTF16Decode (lead, trail)

10.2 Types of Source Code
10.2.1 Strict Mode Code
10.2.2 Non-ECMAScript Functions

11 ECMAScript Language: Lexical Grammar

11.1 Unicode Format-Control Characters
11.2 White Space
11.3 Line Terminators
11.4 Comments
11.5 Tokens

11.6 Names and Keywords
11.6.1 Identifier Names
11.6.1.1 SS: Early Errors
11.6.1.2 SS: StringValue
11.6.2 Reserved Words
11.6.2.1 Keywords
11.6.2.2 Future Reserved Words

11.7 Punctuators

11.8 Literals
11.8.1 Null Literals
11.8.2 Boolean Literals
11.8.3 Numeric Literals
11.8.3.1 SS: MV
11.8.4 String Literals
11.8.4.1 SS: Early Errors
11.8.4.2 SS: StringValue
11.8.4.3 SS: SV
11.8.5 Regular Expression Literals
  11.8.5.1 SS: Early Errors
  11.8.5.2 SS: BodyText
  11.8.5.3 SS: FlagText
11.8.6 Template Literal Lexical Components
  11.8.6.1 SS: TV and TRV
11.9 Automatic Semicolon Insertion
  11.9.1 Rules of Automatic Semicolon Insertion
  11.9.2 Examples of Automatic Semicolon Insertion
12 ECMAScript Language: Expressions
  12.1 Identifiers
    12.1.1 SS: Early Errors
    12.1.2 SS: BoundNames
    12.1.3 SS: IsValidSimpleAssignmentTarget
    12.1.4 SS: StringValue
    12.1.5 RS: BindingInitialization
      12.1.5.1 RS: InitializeBoundName ( name, value, environment )
    12.1.6 RS: Evaluation
  12.2 Primary Expression
    12.2.1 Semantics
      12.2.1.1 SS: CoveredParenthesizedExpression
      12.2.1.2 SS: HasName
      12.2.1.3 SS: IsFunctionDefinition
      12.2.1.4 SS: IsIdentifierRef
      12.2.1.5 SS: IsValidSimpleAssignmentTarget
    12.2.2 The this Keyword
      12.2.2.1 RS: Evaluation
    12.2.3 Identifier Reference
    12.2.4 Literals
      12.2.4.1 RS: Evaluation
    12.2.5 Array Initializer
      12.2.5.1 SS: ElisionWidth
      12.2.5.2 RS: ArrayAccumulation
      12.2.5.3 RS: Evaluation
    12.2.6 Object Initializer
      12.2.6.1 SS: Early Errors
      12.2.6.2 SS: ComputedPropertyContains
12.2.6.3 SS: Contains
12.2.6.4 SS: IsComputedPropertyKey
12.2.6.5 SS: PropName
12.2.6.6 SS: PropertyNameList
12.2.6.7 RS: Evaluation
12.2.6.8 RS: PropertyDefinitionEvaluation
12.2.7 Function Defining Expressions
12.2.8 Regular Expression Literals
  12.2.8.1 SS: Early Errors
  12.2.8.2 RS: Evaluation
12.2.9 Template Literals
  12.2.9.1 SS: TemplateStrings
  12.2.9.2 RS: ArgumentListEvaluation
  12.2.9.3 RS: GetTemplateObject (templateLiteral)
  12.2.9.4 RS: SubstitutionEvaluation
  12.2.9.5 RS: Evaluation
12.2.10 The Grouping Operator
  12.2.10.1 SS: Early Errors
  12.2.10.2 SS: IsFunctionDefinition
  12.2.10.3 SS: IsValidSimpleAssignmentTarget
  12.2.10.4 RS: Evaluation
12.3 Left-Hand-Side Expressions
12.3.1 Static Semantics
  12.3.1.1 SS: CoveredCallExpression
  12.3.1.2 SS: Contains
  12.3.1.3 SS: IsFunctionDefinition
  12.3.1.4 SS: IsDestructuring
  12.3.1.5 SS: IsIdentifierRef
  12.3.1.6 SS: IsValidSimpleAssignmentTarget
12.3.2 Property Accessors
  12.3.2.1 RS: Evaluation
12.3.3 The new Operator
  12.3.3.1 RS: Evaluation
    12.3.3.1.1 RS: EvaluateNew (constructExpr, arguments)
12.3.4 Function Calls
  12.3.4.1 RS: Evaluation
  12.3.4.2 RS: EvaluateCall (ref, arguments, tailPosition)
  12.3.4.3 RS: EvaluateDirectCall (func, thisValue, arguments, tailPosition)
12.3.5 The super Keyword
  12.3.5.1 RS: Evaluation
12.3.5.2 RS: GetSuperConstructor ()
12.3.5.3 RS: MakeSuperPropertyReference (propertyKey, strict)

12.3.6 Argument Lists
12.3.6.1 RS: ArgumentListEvaluation

12.3.7 Tagged Templates
12.3.7.1 RS: Evaluation

12.3.8 Meta Properties
12.3.8.1 RS: Evaluation

12.4 Update Expressions
12.4.1 SS: Early Errors
12.4.2 SS: IsFunctionDefinition
12.4.3 SS: IsValidSimpleAssignmentTarget
12.4.4 Postfix Increment Operator
12.4.4.1 RS: Evaluation
12.4.5 Postfix Decrement Operator
12.4.5.1 RS: Evaluation
12.4.6 Prefix Increment Operator
12.4.6.1 RS: Evaluation
12.4.7 Prefix Decrement Operator
12.4.7.1 RS: Evaluation

12.5 Unary Operators
12.5.1 SS: IsFunctionDefinition
12.5.2 SS: IsValidSimpleAssignmentTarget
12.5.3 The delete Operator
12.5.3.1 SS: Early Errors
12.5.3.2 RS: Evaluation
12.5.4 The void Operator
12.5.4.1 RS: Evaluation
12.5.5 The typeof Operator
12.5.5.1 RS: Evaluation
12.5.6 Unary + Operator
12.5.6.1 RS: Evaluation
12.5.7 Unary - Operator
12.5.7.1 RS: Evaluation
12.5.8 Bitwise NOT Operator (~)
12.5.8.1 RS: Evaluation
12.5.9 Logical NOT Operator (!)
12.5.9.1 RS: Evaluation

12.6 Exponentiation Operator
12.6.1 SS: IsFunctionDefinition
12.6.2 SS: IsValidSimpleAssignmentTarget
12.6.3 RS: Evaluation
12.6.4 Applying the ** Operator

12.7 Multiplicative Operators
12.7.1 SS: IsFunctionDefinition
12.7.2 SS: IsValidSimpleAssignmentTarget
12.7.3 RS: Evaluation
  12.7.3.1 Applying the * Operator
  12.7.3.2 Applying the / Operator
  12.7.3.3 Applying the % Operator

12.8 Additive Operators
12.8.1 SS: IsFunctionDefinition
12.8.2 SS: IsValidSimpleAssignmentTarget
12.8.3 The Addition Operator (+)
  12.8.3.1 RS: Evaluation
12.8.4 The Subtraction Operator (-)
  12.8.4.1 RS: Evaluation
12.8.5 Applying the Additive Operators to Numbers

12.9 Bitwise Shift Operators
12.9.1 SS: IsFunctionDefinition
12.9.2 SS: IsValidSimpleAssignmentTarget
12.9.3 The Left Shift Operator (<<)
  12.9.3.1 RS: Evaluation
12.9.4 The Signed Right Shift Operator (>>)
  12.9.4.1 RS: Evaluation
12.9.5 The Unsigned Right Shift Operator (>>>)
  12.9.5.1 RS: Evaluation

12.10 Relational Operators
12.10.1 SS: IsFunctionDefinition
12.10.2 SS: IsValidSimpleAssignmentTarget
12.10.3 RS: Evaluation
12.10.4 RS: InstanceOfOperator (O, C)

12.11 Equality Operators
12.11.1 SS: IsFunctionDefinition
12.11.2 SS: IsValidSimpleAssignmentTarget
12.11.3 RS: Evaluation

12.12 Binary Bitwise Operators
12.12.1 SS: IsFunctionDefinition
12.12.2 SS: IsValidSimpleAssignmentTarget
12.12.3 RS: Evaluation
12.13 Binary Logical Operators
   12.13.1 SS: IsFunctionDefinition
   12.13.2 SS: IsValidSimpleAssignmentTarget
   12.13.3 RS: Evaluation

   12.14.1 SS: IsFunctionDefinition
   12.14.2 SS: IsValidSimpleAssignmentTarget
   12.14.3 RS: Evaluation

12.15 Assignment Operators
   12.15.1 SS: Early Errors
   12.15.2 SS: IsFunctionDefinition
   12.15.3 SS: IsValidSimpleAssignmentTarget
   12.15.4 RS: Evaluation
   12.15.5 Destructuring Assignment
      12.15.5.1 SS: Early Errors
      12.15.5.2 RS: DestructuringAssignmentEvaluation
      12.15.5.3 RS: IteratorDestructuringAssignmentEvaluation
      12.15.5.4 RS: KeyedDestructuringAssignmentEvaluation

12.16 Comma Operator ( , )
   12.16.1 SS: IsFunctionDefinition
   12.16.2 SS: IsValidSimpleAssignmentTarget
   12.16.3 RS: Evaluation

13 ECMAScript Language: Statements and Declarations

13.1 Statement Semantics
   13.1.1 SS: ContainsDuplicateLabels
   13.1.2 SS: ContainsUndefinedBreakTarget
   13.1.3 SS: ContainsUndefinedContinueTarget
   13.1.4 SS: DeclarationPart
   13.1.5 SS: VarDeclaredNames
   13.1.6 SS: VarScopedDeclarations
   13.1.7 RS: LabelledEvaluation
   13.1.8 RS: Evaluation

13.2 Block
   13.2.1 SS: Early Errors
   13.2.2 SS: ContainsDuplicateLabels
   13.2.3 SS: ContainsUndefinedBreakTarget
   13.2.4 SS: ContainsUndefinedContinueTarget
   13.2.5 SS: LexicallyDeclaredNames
   13.2.6 SS: LexicallyScopedDeclarations
   13.2.7 SS: TopLevelLexicallyDeclaredNames
13.2.8 SS: TopLevelLexicallyScopedDeclarations
13.2.9 SS: TopLevelVarDeclaredNames
13.2.10 SS: TopLevelVarScopedDeclarations
13.2.11 SS: VarDeclaredNames
13.2.12 SS: VarScopedDeclarations
13.2.13 RS: Evaluation
13.2.14 RS: BlockDeclarationInstantiation( code, env )

13.3 Declarations and the Variable Statement
13.3.1 Let and Const Declarations
   13.3.1.1 SS: Early Errors
   13.3.1.2 SS: BoundNames
   13.3.1.3 SS: IsConstantDeclaration
   13.3.1.4 RS: Evaluation
13.3.2 Variable Statement
   13.3.2.1 SS: BoundNames
   13.3.2.2 SS: VarDeclaredNames
   13.3.2.3 SS: VarScopedDeclarations
   13.3.2.4 RS: Evaluation
13.3.3 Destructuring Binding Patterns
   13.3.3.1 SS: BoundNames
   13.3.3.2 SS: ContainsExpression
   13.3.3.3 SS: HasInitializer
   13.3.3.4 SS: IsSimpleParameterList
   13.3.3.5 RS: BindingInitialization
   13.3.3.6 RS: IteratorBindingInitialization
   13.3.3.7 RS: KeyedBindingInitialization

13.4 Empty Statement
   13.4.1 RS: Evaluation

13.5 Expression Statement
   13.5.1 RS: Evaluation

13.6 The if Statement
   13.6.1 SS: Early Errors
   13.6.2 SS: ContainsDuplicateLabels
   13.6.3 SS: ContainsUndefinedBreakTarget
   13.6.4 SS: ContainsUndefinedContinueTarget
   13.6.5 SS: VarDeclaredNames
   13.6.6 SS: VarScopedDeclarations
   13.6.7 RS: Evaluation

13.7 Iteration Statements
   13.7.1 Semantics
13.7.1.1 SS: Early Errors
13.7.1.2 RS: LoopContinues (completion, labelSet)

13.7.2 The **do-while** Statement
13.7.2.1 SS: ContainsDuplicateLabels
13.7.2.2 SS: ContainsUndefinedBreakTarget
13.7.2.3 SS: ContainsUndefinedContinueTarget
13.7.2.4 SS: VarDeclaredNames
13.7.2.5 SS: VarScopedDeclarations
13.7.2.6 RS: LabelledEvaluation

13.7.3 The **while** Statement
13.7.3.1 SS: ContainsDuplicateLabels
13.7.3.2 SS: ContainsUndefinedBreakTarget
13.7.3.3 SS: ContainsUndefinedContinueTarget
13.7.3.4 SS: VarDeclaredNames
13.7.3.5 SS: VarScopedDeclarations
13.7.3.6 RS: LabelledEvaluation

13.7.4 The **for** Statement
13.7.4.1 SS: Early Errors
13.7.4.2 SS: ContainsDuplicateLabels
13.7.4.3 SS: ContainsUndefinedBreakTarget
13.7.4.4 SS: ContainsUndefinedContinueTarget
13.7.4.5 SS: VarDeclaredNames
13.7.4.6 SS: VarScopedDeclarations
13.7.4.7 RS: LabelledEvaluation
13.7.4.8 RS: ForBodyEvaluation(test, increment, stmt, perIterationBindings, labelSet)
13.7.4.9 RS: CreatePerIterationEnvironment(perIterationBindings)

13.7.5 The **for-in** and **for-of** Statements
13.7.5.1 SS: Early Errors
13.7.5.2 SS: BoundNames
13.7.5.3 SS: ContainsDuplicateLabels
13.7.5.4 SS: ContainsUndefinedBreakTarget
13.7.5.5 SS: ContainsUndefinedContinueTarget
13.7.5.6 SS: IsDestructuring
13.7.5.7 SS: VarDeclaredNames
13.7.5.8 SS: VarScopedDeclarations
13.7.5.9 RS: BindingInitialization
13.7.5.10 RS: BindingInstantiation
13.7.5.11 RS: LabelledEvaluation
13.7.5.12 RS: ForIn/OfHeadEvaluation(TDZnames, expr, iterationKind)
13.7.5.13 RS: ForIn/OfBodyEvaluation(lhs, stmt, iterator, iterationKind, lhsKind, labelSet)
13.8 The `continue` Statement
   13.8.1 SS: Early Errors
   13.8.2 SS: ContainsUndefinedContinueTarget
   13.8.3 RS: Evaluation

13.9 The `break` Statement
   13.9.1 SS: Early Errors
   13.9.2 SS: ContainsUndefinedBreakTarget
   13.9.3 RS: Evaluation

13.10 The `return` Statement
   13.10.1 RS: Evaluation

13.11 The `with` Statement
   13.11.1 SS: Early Errors
   13.11.2 SS: ContainsDuplicateLabels
   13.11.3 SS: ContainsUndefinedBreakTarget
   13.11.4 SS: ContainsUndefinedContinueTarget
   13.11.5 SS: VarDeclaredNames
   13.11.6 SS: VarScopedDeclarations
   13.11.7 RS: Evaluation

13.12 The `switch` Statement
   13.12.1 SS: Early Errors
   13.12.2 SS: ContainsDuplicateLabels
   13.12.3 SS: ContainsUndefinedBreakTarget
   13.12.4 SS: ContainsUndefinedContinueTarget
   13.12.5 SS: LexicallyDeclaredNames
   13.12.6 SS: LexicallyScopedDeclarations
   13.12.7 SS: VarDeclaredNames
   13.12.8 SS: VarScopedDeclarations
   13.12.9 RS: CaseBlockEvaluation
   13.12.10 RS: CaseSelectorEvaluation
   13.12.11 RS: Evaluation

13.13 Labelled Statements
   13.13.1 SS: Early Errors
   13.13.2 SS: ContainsDuplicateLabels
   13.13.3 SS: ContainsUndefinedBreakTarget
   13.13.4 SS: ContainsUndefinedContinueTarget
   13.13.5 SS: IsLabelledFunction ( `stmt` )
   13.13.6 SS: LexicallyDeclaredNames
   13.13.7 SS: LexicallyScopedDeclarations
13.13.8 SS: TopLevelLexicallyDeclaredNames
13.13.9 SS: TopLevelLexicallyScopedDeclarations
13.13.10 SS: TopLevelVarDeclaredNames
13.13.11 SS: TopLevelVarScopedDeclarations
13.13.12 SS: VarDeclaredNames
13.13.13 SS: VarScopedDeclarations
13.13.14 RS: LabelledEvaluation
13.13.15 RS: Evaluation
13.14 The **throw** Statement

13.15 The **try** Statement

13.15.1 SS: Early Errors
13.15.2 SS: ContainsDuplicateLabels
13.15.3 SS: ContainsUndefinedBreakTarget
13.15.4 SS: ContainsUndefinedContinueTarget
13.15.5 SS: VarDeclaredNames
13.15.6 SS: VarScopedDeclarations
13.15.7 RS: CatchClauseEvaluation
13.15.8 RS: Evaluation
13.16 The **debugger** Statement

13.16.1 RS: Evaluation

14 ECMAScript Language: Functions and Classes

14.1 Function Definitions


14.1.2 SS: Early Errors
14.1.3 SS: BoundNames
14.1.4 SS: Contains
14.1.5 SS: ContainsExpression
14.1.6 SS: ContainsUseStrict
14.1.7 SS: ExpectedArgumentCount
14.1.8 SS: HasInitializer
14.1.9 SS: HasName
14.1.10 SS: IsAnonymousFunctionDefinition (expr)
14.1.11 SS: IsConstantDeclaration
14.1.12 SS: IsFunctionDefinition
14.1.13 SS: IsSimpleParameterList
14.1.14 SS: LexicallyDeclaredNames
14.1.15 SS: LexicallyScopedDeclarations
14.1.16 SS: VarDeclaredNames
14.1.17 SS: VarScopedDeclarations
14.1.18 RS: EvaluateBody
14.1.19 RS: IteratorBindingInitialization
14.1.20 RS: InstantiateFunctionObject
14.1.21 RS: Evaluation

14.2 Arrow Function Definitions
14.2.1 SS: Early Errors
14.2.2 SS: BoundNames
14.2.3 SS: Contains
14.2.4 SS: ContainsExpression
14.2.5 SS: ContainsUseStrict
14.2.6 SS: ExpectedArgumentCount
14.2.7 SS: HasName
14.2.8 SS: IsSimpleParameterList
14.2.9 SS: CoveredFormalsList
14.2.10 SS: LexicallyDeclaredNames
14.2.11 SS: LexicallyScopedDeclarations
14.2.12 SS: VarDeclaredNames
14.2.13 SS: VarScopedDeclarations
14.2.14 RS: IteratorBindingInitialization
14.2.15 RS: EvaluateBody
14.2.16 RS: Evaluation

14.3 Method Definitions
14.3.1 SS: Early Errors
14.3.2 SS: ComputedPropertyContains
14.3.3 SS: ExpectedArgumentCount
14.3.4 SS: HasDirectSuper
14.3.5 SS: PropName
14.3.6 SS: SpecialMethod
14.3.7 RS: DefineMethod
14.3.8 RS: PropertyDefinitionEvaluation

14.4 Generator Function Definitions
14.4.1 SS: Early Errors
14.4.2 SS: BoundNames
14.4.3 SS: ComputedPropertyContains
14.4.4 SS: Contains
14.4.5 SS: HasDirectSuper
14.4.6 SS: HasName
14.4.7 SS: IsConstantDeclaration
14.4.8 SS: IsFunctionDefinition
14.4.9 SS: PropName
14.4.10 RS: EvaluateBody
14.4.11 RS: InstantiateFunctionObject
14.4.12 RS: PropertyDefinitionEvaluation
14.4.13 RS: Evaluation

14.5 Class Definitions
14.5.1 SS: Early Errors
14.5.2 SS: BoundNames
14.5.3 SS: ConstructorMethod
14.5.4 SS: Contains
14.5.5 SS: ComputedPropertyContains
14.5.6 SS: HasName
14.5.7 SS: IsConstantDeclaration
14.5.8 SS: IsFunctionDefinition
14.5.9 SS: IsStatic
14.5.10 SS: NonConstructorMethodDefinitions
14.5.11 SS: PrototypePropertyNameList
14.5.12 SS: PropName
14.5.13 RS: ClassDefinitionEvaluation
14.5.14 RS: BindingClassDeclarationEvaluation
14.5.15 RS: Evaluation

14.6 Async Function Definitions
14.6.1 SS: Early Errors
14.6.2 SS: BoundNames
14.6.3 SS: ComputedPropertyContains
14.6.4 SS: Contains
14.6.5 SS: HasDirectSuper
14.6.6 SS: HasName
14.6.7 SS: IsConstantDeclaration
14.6.8 SS: IsFunctionDefinition
14.6.9 SS: PropName
14.6.10 RS: InstantiateFunctionObject
14.6.11 RS: EvaluateBody
14.6.12 RS: PropertyDefinitionEvaluation
14.6.13 RS: Evaluation

14.7 Async Arrow Function Definitions
14.7.1 SS: Early Errors
14.7.2 SS: CoveredAsyncArrowHead
14.7.3 SS: BoundNames
14.7.4 SS: Contains
14.7.5 SS: ContainsExpression
14.7.6 SS: ExpectedArgumentCount
14.7.7 SS: HasName
14.7.8 SS: IsSimpleParameterList
14.7.9 SS: LexicallyDeclaredNames
14.7.10 SS: LexicallyScopedDeclarations
14.7.11 SS: VarDeclaredNames
14.7.12 SS: VarScopedDeclarations
14.7.13 RS: IteratorBindingInitialization
14.7.14 RS: EvaluateBody
14.7.15 RS: Evaluation

14.8 Tail Position Calls
14.8.1 SS: IsInTailPosition( call )
14.8.2 SS: HasCallInTailPosition
   14.8.2.1 Statement Rules
   14.8.2.2 Expression Rules
14.8.3 RS: PrepareForTailCall ( )

15 ECMAScript Language: Scripts and Modules

15.1 Scripts
   15.1.1 SS: Early Errors
   15.1.2 SS: IsStrict
   15.1.3 SS: LexicallyDeclaredNames
   15.1.4 SS: LexicallyScopedDeclarations
   15.1.5 SS: VarDeclaredNames
   15.1.6 SS: VarScopedDeclarations
   15.1.7 RS: Evaluation
   15.1.8 Script Records
   15.1.9 ParseScript ( sourceText, realm, hostDefined )
   15.1.10 ScriptEvaluation ( scriptRecord )
   15.1.11 RS: GlobalDeclarationInstantiation ( script, env )
   15.1.12 RS: ScriptEvaluationJob ( sourceText, hostDefined )

15.2 Modules
   15.2.1 Module Semantics
      15.2.1.1 SS: Early Errors
      15.2.1.2 SS: ContainsDuplicateLabels
      15.2.1.3 SS: ContainsUndefinedBreakTarget
      15.2.1.4 SS: ContainsUndefinedContinueTarget
      15.2.1.5 SS: ExportedBindings
      15.2.1.6 SS: ExportedNames
      15.2.1.7 SS: ExportEntries
      15.2.1.8 SS: ImportEntries
15.2.19 SS: ImportedLocalNames ( importEntries )
15.2.1.10 SS: ModuleRequests
15.2.1.11 SS: LexicallyDeclaredNames
15.2.1.12 SS: LexicallyScopedDeclarations
15.2.1.13 SS: VarDeclaredNames
15.2.1.14 SS: VarScopedDeclarations
15.2.1.15 Abstract Module Records
15.2.1.16 Source Text Module Records
   15.2.1.16.1 ParseModule ( sourceText, realm, hostDefined )
   15.2.1.16.2 GetExportedNames( exportStarSet ) Concrete Method
   15.2.1.16.3 ResolveExport( exportName, resolveSet ) Concrete Method
   15.2.1.16.4 ModuleDeclarationInstantiation( ) Concrete Method
   15.2.1.16.5 ModuleEvaluation( ) Concrete Method
15.2.1.17 RS: HostResolveImportedModule ( referencingModule, specifier )
15.2.1.18 RS: GetModuleNameSpace( module )
15.2.1.19 RS: TopLevelModuleEvaluationJob ( sourceText, hostDefined )
15.2.1.20 RS: Evaluation
15.2.2 Imports
   15.2.2.1 SS: Early Errors
   15.2.2.2 SS: BoundNames
   15.2.2.3 SS: ImportEntries
   15.2.2.4 SS: ImportEntriesForModule
   15.2.2.5 SS: ModuleRequests
15.2.3 Exports
   15.2.3.1 SS: Early Errors
   15.2.3.2 SS: BoundNames
   15.2.3.3 SS: ExportedBindings
   15.2.3.4 SS: ExportedNames
   15.2.3.5 SS: ExportEntries
   15.2.3.6 SS: ExportEntriesForModule
   15.2.3.7 SS: IsConstantDeclaration
   15.2.3.8 SS: LexicallyScopedDeclarations
   15.2.3.9 SS: ModuleRequests
   15.2.3.10 SS: ReferencedBindings
   15.2.3.11 RS: Evaluation
16 Error Handling and Language Extensions
   16.1 HostReportErrors ( errorList )
16.2 Forbidden Extensions
17 ECMAScript Standard Built-in Objects
18 The Global Object
18.1 Value Properties of the Global Object
   18.1.1 Infinity
   18.1.2 NaN
   18.1.3 undefined

18.2 Function Properties of the Global Object
   18.2.1 eval( x )
      18.2.1.1 RS: PerformEval( x, evalRealm, strictCaller, direct )
         18.2.1.1.1 Additional Early Error Rules for Eval Outside Functions
         18.2.1.1.2 Additional Early Error Rules for Eval Outside Methods
         18.2.1.1.3 Additional Early Error Rules for Eval Outside Constructor Methods
      18.2.1.2 HostEnsureCanCompileStrings( callerRealm, calleeRealm )
      18.2.1.3 RS: EvalDeclarationInstantiation( body, varEnv, lexEnv, strict )
   18.2.2 isFinite( number )
   18.2.3 isNaN( number )
   18.2.4 parseFloat( string )
   18.2.5 parseInt( string, radix )

18.2.6 URI Handling Functions
   18.2.6.1 URI Syntax and Semantics
      18.2.6.1.1 RS: Encode( string, unescapedSet )
      18.2.6.1.2 RS: Decode( string, reservedSet )
   18.2.6.2 decodeURI( encodedURI )
   18.2.6.3 decodeURIComponent( encodedURIComponent )
   18.2.6.4 encodeURI( uri )
   18.2.6.5 encodeURIComponent( uriComponent )

18.3 Constructor Properties of the Global Object
   18.3.1 Array( … )
   18.3.2 ArrayBuffer( … )
   18.3.3 Boolean( … )
   18.3.4 DataView( … )
   18.3.5 Date( … )
   18.3.6 Error( … )
   18.3.7 EvalError( … )
   18.3.8 Float32Array( … )
   18.3.9 Float64Array( … )
   18.3.10 Function( … )
   18.3.11 Int8Array( … )
   18.3.12 Int16Array( … )
   18.3.13 Int32Array( … )
   18.3.14 Map( … )
   18.3.15 Number( … )
18.3.16 Object (…)
18.3.17 Proxy (…)
18.3.18 Promise (…)
18.3.19 RangeError (…)
18.3.20 ReferenceError (…)
18.3.21 RegExp (…)
18.3.22 Set (…)
18.3.23 SharedArrayBuffer (…)
18.3.24 String (…)
18.3.25 Symbol (…)
18.3.26 SyntaxError (…)
18.3.27 TypeError (…)
18.3.28 Uint8Array (…)
18.3.29 Uint8ClampedArray (…)
18.3.30 Uint16Array (…)
18.3.31 Uint32Array (…)
18.3.32 URIError (…)
18.3.33 WeakMap (…)
18.3.34 WeakSet (…)

18.4 Other Properties of the Global Object
18.4.1 Atomics
18.4.2 JSON
18.4.3 Math
18.4.4 Reflect

19 Fundamental Objects
19.1 Object Objects
19.1.1 The Object Constructor
   19.1.1.1 Object ( [ value ] )
19.1.2 Properties of the Object Constructor
   19.1.2.1 Object.assign ( target, ...sources )
   19.1.2.2 Object.create ( O, Properties )
   19.1.2.3 Object.defineProperties ( O, Properties )
      19.1.2.3.1 RS: ObjectDefineProperties ( O, Properties )
   19.1.2.4 Object.defineProperty ( O, P, Attributes )
   19.1.2.5 Object.entries ( O )
   19.1.2.6 Object.freeze ( O )
   19.1.2.7 Object.getOwnPropertyDescriptor ( O, P )
   19.1.2.8 Object.getOwnPropertyDescriptors ( O )
   19.1.2.9 Object.getOwnPropertyNames ( O )
   19.1.2.10 Object.getOwnPropertySymbols ( O )
19.1.2.10.1 RS: GetOwnPropertyKeys (O, Type)
19.1.2.11 Object.getPrototypeOf (O)
19.1.2.12 Object.is (value1, value2)
19.1.2.13 Object.isExtensible (O)
19.1.2.14 Object.isFrozen (O)
19.1.2.15 Object.isSealed (O)
19.1.2.16 Object.keys (O)
19.1.2.17 Object.preventExtensions (O)
19.1.2.18 Object.prototype
19.1.2.19 Object.seal (O)
19.1.2.20 Object.setPrototypeOf (O, proto)
19.1.2.21 Object.values (O)

19.1.3 Properties of the Object Prototype Object
19.1.3.1 Object.prototype.constructor
19.1.3.2 Object.prototype.hasOwnProperty (V)
19.1.3.3 Object.prototype.isPrototypeOf (V)
19.1.3.4 Object.prototype.propertyIsEnumerable (V)
19.1.3.5 Object.prototype.toLocaleString ([reserved1 [, reserved2 ]])
19.1.3.6 Object.prototype.toString ()
19.1.3.7 Object.prototype.valueOf ()

19.1.4 Properties of Object Instances

19.2 Function Objects
19.2.1 The Function Constructor
19.2.1.1 Function (p1, p2, ..., pn, body)
19.2.1.1.1 RS: CreateDynamicFunction (constructor, newTarget, kind, args)
19.2.2 Properties of the Function Constructor
19.2.2.1 Function.length
19.2.2.2 Function.prototype
19.2.3 Properties of the Function Prototype Object
19.2.3.1 Function.prototype.apply (thisArg, argArray)
19.2.3.2 Function.prototype.bind (thisArg, ...args)
19.2.3.3 Function.prototype.call (thisArg, ...args)
19.2.3.4 Function.prototype.constructor
19.2.3.5 Function.prototype.toString ()
19.2.3.6 Function.prototype @@hasInstance (V)

19.2.4 Function Instances
19.2.4.1 length
19.2.4.2 name
19.2.4.3 prototype

19.3 Boolean Objects
19.3.1 The Boolean Constructor
  19.3.1.1 Boolean (value)
19.3.2 Properties of the Boolean Constructor
  19.3.2.1 Boolean.prototype
19.3.3 Properties of the Boolean Prototype Object
  19.3.3.1 Boolean.prototype.constructor
  19.3.3.2 Boolean.prototype.toString()
  19.3.3.3 Boolean.prototype.valueOf()
19.3.4 Properties of Boolean Instances

19.4 Symbol Objects
  19.4.1 The Symbol Constructor
    19.4.1.1 Symbol ([description])
  19.4.2 Properties of the Symbol Constructor
    19.4.2.1 Symbol.for (key)
    19.4.2.2 Symbol.hasInstance
    19.4.2.3 Symbol.isConcatSpreadable
    19.4.2.4 Symbol.iterator
    19.4.2.5 Symbol.keyFor (sym)
    19.4.2.6 Symbol.match
    19.4.2.7 Symbol.prototype
    19.4.2.8 Symbol.replace
    19.4.2.9 Symbol.search
    19.4.2.10 Symbol.species
    19.4.2.11 Symbol.split
    19.4.2.12 Symbol.toPrimitive
    19.4.2.13 Symbol.toStringTag
    19.4.2.14 Symbol.unscopables
  19.4.3 Properties of the Symbol Prototype Object
    19.4.3.1 Symbol.prototype.constructor
    19.4.3.2 Symbol.prototype.toString()
      19.4.3.2.1 RS: SymbolDescriptiveString (sym)
    19.4.3.3 Symbol.prototype.valueOf()
    19.4.3.4 Symbol.prototype [@@toPrimitive] (hint)
    19.4.3.5 Symbol.prototype [@@toStringTag]
  19.4.4 Properties of Symbol Instances

19.5 Error Objects
  19.5.1 The Error Constructor
    19.5.1.1 Error (message)
  19.5.2 Properties of the Error Constructor
    19.5.2.1 Error.prototype
19.5.3 Properties of the Error Prototype Object
   19.5.3.1 Error.prototype.constructor
   19.5.3.2 Error.prototype.message
   19.5.3.3 Error.prototype.name
   19.5.3.4 Error.prototype.toString()
19.5.4 Properties of Error Instances
19.5.5 Native Error Types Used in This Standard
   19.5.5.1 EvalError
   19.5.5.2 RangeError
   19.5.5.3 ReferenceError
   19.5.5.4 SyntaxError
   19.5.5.5 TypeError
   19.5.5.6 URIError
19.5.6 NativeError Object Structure
   19.5.6.1 NativeError Constructors
      19.5.6.1.1 NativeError ( message )
   19.5.6.2 Properties of the NativeError Constructors
      19.5.6.2.1 NativeError.prototype
   19.5.6.3 Properties of the NativeError Prototype Objects
      19.5.6.3.1 NativeError.prototype.constructor
      19.5.6.3.2 NativeError.prototype.message
      19.5.6.3.3 NativeError.prototype.name
   19.5.6.4 Properties of NativeError Instances

20 Numbers and Dates
20.1 Number Objects
   20.1.1 The Number Constructor
      20.1.1.1 Number ( value )
   20.1.2 Properties of the Number Constructor
      20.1.2.1 Number.EPSILON
      20.1.2.2 Number.isFinite ( number )
      20.1.2.3 Number.isInteger ( number )
      20.1.2.4 Number.isNaN ( number )
      20.1.2.5 Number.isSafeInteger ( number )
      20.1.2.6 Number.MAX_SAFE_INTEGER
      20.1.2.7 Number.MAX_VALUE
      20.1.2.8 Number.MIN_SAFE_INTEGER
      20.1.2.9 Number.MIN_VALUE
      20.1.2.10 Number.NaN
      20.1.2.11 Number.NEGATIVE_INFINITY
      20.1.2.12 Number.parseFloat ( string )
20.1.2.13 `Number.parseInt(string, radix)`
20.1.2.14 `Number.POSITIVE_INFINITY`
20.1.2.15 `Number.prototype`

20.1.3 Properties of the Number Prototype Object

20.1.3.1 `Number.prototype.constructor`
20.1.3.2 `Number.prototype.toExponential(fractionDigits)`
20.1.3.3 `Number.prototype.toFixed(fractionDigits)`
20.1.3.4 `Number.prototype.toLocaleString([reserved1, reserved2])`
20.1.3.5 `Number.prototype.toPrecision(precision)`
20.1.3.6 `Number.prototype.toString([radix])`
20.1.3.7 `Number.prototype.valueOf()`

20.1.4 Properties of Number Instances

20.2 The Math Object

20.2.1 Value Properties of the Math Object

20.2.1.1 `Math.E`
20.2.1.2 `Math.LN10`
20.2.1.3 `Math.LN2`
20.2.1.4 `Math.LOG10E`
20.2.1.5 `Math.LOG2E`
20.2.1.6 `Math.PI`
20.2.1.7 `Math.SQRT1_2`
20.2.1.8 `Math.SQRT2`
20.2.1.9 `Math @@toStringTag`

20.2.2 Function Properties of the Math Object

20.2.2.1 `Math.abs(x)`
20.2.2.2 `Math.acos(x)`
20.2.2.3 `Math.acosh(x)`
20.2.2.4 `Math.asin(x)`
20.2.2.5 `Math.asinh(x)`
20.2.2.6 `Math.atan(x)`
20.2.2.7 `Math.atanh(x)`
20.2.2.8 `Math.atan2(y, x)`
20.2.2.9 `Math.cbrt(x)`
20.2.2.10 `Math.ceil(x)`
20.2.2.11 `Math.clz32(x)`
20.2.2.12 `Math.cos(x)`
20.2.2.13 `Math.cosh(x)`
20.2.2.14 `Math.exp(x)`
20.2.2.15 `Math.expm1(x)`
20.2.2.16 `Math.floor(x)`
20.2.2.17 Math.fround (x)
20.2.2.18 Math.hypot (value1, value2,...values)
20.2.2.19 Math.imul (x, y)
20.2.2.20 Math.log (x)
20.2.2.21 Math.log1p (x)
20.2.2.22 Math.log10 (x)
20.2.2.23 Math.log2 (x)
20.2.2.24 Math.max (value1, value2,...values)
20.2.2.25 Math.min (value1, value2,...values)
20.2.2.26 Math.pow (base, exponent)
20.2.2.27 Math.random ()
20.2.2.28 Math.round (x)
20.2.2.29 Math.sign (x)
20.2.2.30 Math.sin (x)
20.2.2.31 Math.sinh (x)
20.2.2.32 Math.sqrt (x)
20.2.2.33 Math.tan (x)
20.2.2.34 Math.tanh (x)
20.2.2.35 Math.trunc (x)

20.3 Date Objects

20.3.1 Overview of Date Objects and Definitions of Abstract Operations
20.3.1.1 Time Values and Time Range
20.3.1.2 Day Number and Time within Day
20.3.1.3 Year Number
20.3.1.4 Month Number
20.3.1.5 Date Number
20.3.1.6 Week Day
20.3.1.7 Local Time Zone Adjustment
20.3.1.8 Daylight Saving Time Adjustment
20.3.1.9 LocalTime (t)
20.3.1.10 UTC (t)
20.3.1.11 Hours, Minutes, Second, and Milliseconds
20.3.1.12 MakeTime (hour, min, sec, ms)
20.3.1.13 MakeDay (year, month, date)
20.3.1.14 MakeDate (day, time)
20.3.1.15 TimeClip (time)
20.3.1.16 Date Time String Format
20.3.1.16.1 Extended Years

20.3.2 The Date Constructor
20.3.2.1 Date (year, month [, date [, hours [, minutes [, seconds [, ms]]]]]])
20.3.2.2 Date ( value )
20.3.2.3 Date ( )

20.3.3 Properties of the Date Constructor
20.3.3.1 Date.now ( )
20.3.3.2 Date.parse ( string )
20.3.3.3 Date.prototype
20.3.3.4 Date.UTC ( year [ , month [ , date [ , hours [ , minutes [ , seconds [ , ms ] ] ] ] ] ] )

20.3.4 Properties of the Date Prototype Object
20.3.4.1 Date.prototype.constructor
20.3.4.2 Date.prototype.getDate ( )
20.3.4.3 Date.prototype.getDay ( )
20.3.4.4 Date.prototype.getFullYear ( )
20.3.4.5 Date.prototype.getHours ( )
20.3.4.6 Date.prototype.getMilliseconds ( )
20.3.4.7 Date.prototype.getMinutes ( )
20.3.4.8 Date.prototype.getMonth ( )
20.3.4.9 Date.prototype.getSeconds ( )
20.3.4.10 Date.prototype.getTime ( )
20.3.4.11 Date.prototype.getTimezoneOffset ( )
20.3.4.12 Date.prototype.getUTCDate ( )
20.3.4.13 Date.prototype.getUTCDay ( )
20.3.4.14 Date.prototype.getUTCFullYear ( )
20.3.4.15 Date.prototype.getUTCHours ( )
20.3.4.16 Date.prototype.getUTCMilliseconds ( )
20.3.4.17 Date.prototype.getUTCMinutes ( )
20.3.4.18 Date.prototype.getUTCMonth ( )
20.3.4.19 Date.prototype.getUTCSeconds ( )
20.3.4.20 Date.prototype.setDate ( date )
20.3.4.21 Date.prototype.setFullYear ( year [ , month [ , date ] ] )
20.3.4.22 Date.prototype.setHours ( hour [ , min [ , sec [ , ms ] ] ] )
20.3.4.23 Date.prototype.setMilliseconds ( ms )
20.3.4.24 Date.prototype.setMinutes ( min [ , sec [ , ms ] ] )
20.3.4.25 Date.prototype.setMonth ( month [ , date ] )
20.3.4.26 Date.prototype.setSeconds ( sec [ , ms ] )
20.3.4.27 Date.prototype.setTime ( time )
20.3.4.28 Date.prototype.setUTCDate ( date )
20.3.4.29 Date.prototype.setUTCFullYear ( year [ , month [ , date ] ] )
20.3.4.30 Date.prototype.setUTCHours ( hour [ , min [ , sec [ , ms ] ] ] )
20.3.4.31 Date.prototype.setUTCMilliseconds ( ms )
20.3.4.32 Date.prototype.setUTCMinutes ( min [ , sec [ , ms ] ] )
20.3.4.33 Date.prototype.setUTCMonth ( month [, date ] )
20.3.4.34 Date.prototype.setUTCSeconds ( sec [, ms ] )
20.3.4.35 Date.prototype.toDateString ( )
20.3.4.36 Date.prototype.toISOString ( )
20.3.4.37 Date.prototype.toJSON ( key )
20.3.4.38 Date.prototype.toLocaleDateString ([ reserved1 [, reserved2 ] ])
20.3.4.39 Date.prototype.toLocaleString ([ reserved1 [, reserved2 ] ])
20.3.4.40 Date.prototype.toLocaleTimeString ([ reserved1 [, reserved2 ] ])
20.3.4.41 Date.prototype.toString ( )
  20.3.4.41.1 RS: ToDateString( tv )
20.3.4.42 Date.prototype.toTimeString ( )
20.3.4.43 Date.prototype.toUTCString ( )
20.3.4.44 Date.prototype.valueOf ( )
20.3.4.45 Date.prototype [ @@toPrimitive ] ( hint )

20.3.5 Properties of Date Instances

21 Text Processing

21.1 String Objects

21.1.1 The String Constructor
  21.1.1.1 String ( value )

21.1.2 Properties of the String Constructor
  21.1.2.1 String.fromCharCode ( ...codeUnits )
  21.1.2.2 String.fromCharCode ( ...codePoints )
  21.1.2.3 String.prototype
  21.1.2.4 String.raw ( template, ...substitutions )

21.1.3 Properties of the String Prototype Object
  21.1.3.1 String.prototype.charAt ( pos )
  21.1.3.2 String.prototype.charCodeAtAt ( pos )
  21.1.3.3 String.prototype.codePointAt ( pos )
  21.1.3.4 String.prototype.concat ( ...args )
  21.1.3.5 String.prototype.constructor
  21.1.3.6 String.prototype.endsWith ( searchString [, endPosition ] )
  21.1.3.7 String.prototype.includes ( searchString [, position ] )
  21.1.3.8 String.prototype.indexOf ( searchString [, position ] )
  21.1.3.9 String.prototype.lastIndexOf ( searchString [, position ] )
  21.1.3.10 String.prototype.localeCompare ( that [, reserved1 [, reserved2 ] ] )
  21.1.3.11 String.prototype.match ( rexp )
  21.1.3.12 String.prototype.normalize ([ form ])
  21.1.3.13 String.prototype.padEnd( maxLength [, fillString ] )
  21.1.3.14 String.prototype.padStart( maxLength [, fillString ] )
  21.1.3.15 String.prototype.repeat ( count )
21.1.3.16 String.prototype.replace ( searchValue, replaceValue )
21.1.3.16.1 RS: GetSubstitution ( matched, str, position, captures, replacement )
21.1.3.17 String.prototype.search ( regexp )
21.1.3.18 String.prototype.slice ( start, end )
21.1.3.19 String.prototype.split ( separator, limit )
21.1.3.19.1 RS: SplitMatch ( S, q, R )
21.1.3.20 String.prototype.startsWith ( searchString [, position ] )
21.1.3.21 String.prototype.substring ( start, end )
21.1.3.22 String.prototype.toLocaleLowerCase ( [ reserved1 [, reserved2 ] ] )
21.1.3.23 String.prototype.toLocaleUpperCase ( [ reserved1 [, reserved2 ] ] )
21.1.3.24 String.prototype.toLowerCase ()
21.1.3.25 String.prototype.toString ()
21.1.3.26 String.prototype.toUpperCase ()
21.1.3.27 String.prototype.trim ()
21.1.3.28 String.prototype.valueOf ()
21.1.3.29 String.prototype [ @@iterator ] ()

21.1.4 Properties of String Instances
21.1.4.1 length

21.1.5 String Iterator Objects
21.1.5.1 CreateStringIterator ( string )
21.1.5.2 The %StringIteratorPrototype% Object
21.1.5.2.1 %StringIteratorPrototype%.next ()
21.1.5.2.2 %StringIteratorPrototype% [ @@toStringTag ]
21.1.5.3 Properties of String Iterator Instances

21.2 RegExp (Regular Expression) Objects
21.2.1 Patterns
21.2.1.1 SS: Early Errors
21.2.2 Pattern Semantics
21.2.2.1 Notation
21.2.2.2 Pattern
21.2.2.3 Disjunction
21.2.2.4 Alternative
21.2.2.5 Term
21.2.2.5.1 RS: RepeatMatcher ( m, min, max, greedy, x, c, parenIndex, parenCount )
21.2.2.6 Assertion
21.2.2.6.1 RS: WordCharacters ( )
21.2.2.6.2 RS: IsWordChar ( e )
21.2.2.7 Quantifier
21.2.2.8 Atom
21.2.2.8.1 RS: CharacterSetMatcher ( A, invert )
21.2.2.8.2 RS: Canonicalize (ch)

21.2.2.9 AtomEscape
21.2.2.10 CharacterEscape
21.2.2.11 DecimalEscape
21.2.2.12 CharacterClassEscape
21.2.2.13 CharacterClass
21.2.2.14 ClassRanges
21.2.2.15 NonemptyClassRanges
   21.2.2.15.1 RS: CharacterRange (A, B)
21.2.2.16 NonemptyClassRangesNoDash
21.2.2.17 ClassAtom
21.2.2.18 ClassAtomNoDash
21.2.2.19 ClassEscape

21.2.3 The RegExp Constructor
   21.2.3.1 RegExp (pattern, flags)

21.2.3.2 Abstract Operations for the RegExp Constructor
   21.2.3.2.1 RS: RegExpAlloc (newTarget)
   21.2.3.2.2 RS: RegExpInitialize (obj, pattern, flags)
   21.2.3.2.3 RS: RegExpCreate (P, F)
   21.2.3.2.4 RS: EscapeRegExpPattern (P, F)

21.2.4 Properties of the RegExp Constructor
   21.2.4.1 RegExp.prototype
   21.2.4.2 get RegExp [ @@species ]

21.2.5 Properties of the RegExp Prototype Object
   21.2.5.1 RegExp.prototype.constructor
   21.2.5.2 RegExp.prototype.exec (string)
      21.2.5.2.1 RS: RegExpExec (R, S)
      21.2.5.2.2 RS: RegExpBuiltInExec (R, S)
      21.2.5.2.3 AdvanceStringIndex (S, index, unicode)
   21.2.5.3 get RegExp.prototype.flags
   21.2.5.4 get RegExp.prototype.global
   21.2.5.5 get RegExp.prototype.ignoreCase
   21.2.5.6 RegExp.prototype [ @@match ] (string)
   21.2.5.7 get RegExp.prototype.multiline
   21.2.5.8 RegExp.prototype [ @@replace ] (string, replaceValue)
   21.2.5.9 RegExp.prototype [ @@search ] (string)
   21.2.5.10 get RegExp.prototype.source
   21.2.5.11 RegExp.prototype [ @@split ] (string, limit)
   21.2.5.12 get RegExp.prototype.sticky
   21.2.5.13 RegExp.prototype.test (S)
21.2.5.14 RegExp.prototype.toString ()
21.2.5.15 get RegExp.prototype.unicode
21.2.6 Properties of RegExp Instances
21.2.6.1 lastIndexOf

22 Indexed Collections
22.1 Array Objects
22.1.1 The Array Constructor
22.1.1.1 Array ()
22.1.1.2 Array ( len )
22.1.1.3 Array ( ...items )
22.1.2 Properties of the Array Constructor
22.1.2.1 Array.from ( items [ , mapfn [ , thisArg ] ] )
22.1.2.2 Array.isArray ( arg )
22.1.2.3 Array.of ( ...items )
22.1.2.4 Array.prototype
22.1.2.5 get Array [ @@species ]
22.1.3 Properties of the Array Prototype Object
22.1.3.1 Array.prototype.concat ( ...arguments )
22.1.3.1.1 RS: IsConcatSpreadable ( O )
22.1.3.2 Array.prototype.constructor
22.1.3.3 Array.prototype.copyWithin ( target, start [ , end ] )
22.1.3.4 Array.prototype.entries ( )
22.1.3.5 Array.prototype.every ( callbackfn [ , thisArg ] )
22.1.3.6 Array.prototype.fill ( value [ , start [ , end ] ] )
22.1.3.7 Array.prototype.filter ( callbackfn [ , thisArg ] )
22.1.3.8 Array.prototype.find ( predicate [ , thisArg ] )
22.1.3.9 Array.prototype.findIndex ( predicate [ , thisArg ] )
22.1.3.10 Array.prototype.forEach ( callbackfn [ , thisArg ] )
22.1.3.11 Array.prototype.includes ( searchElement [ , fromIndex ] )
22.1.3.12 Array.prototype.indexOf ( searchElement [ , fromIndex ] )
22.1.3.13 Array.prototype.join ( separator )
22.1.3.14 Array.prototype.keys ( )
22.1.3.15 Array.prototype.lastIndexOf ( searchElement [ , fromIndex ] )
22.1.3.16 Array.prototype.map ( callbackfn [ , thisArg ] )
22.1.3.17 Array.prototype.pop ( )
22.1.3.18 Array.prototype.push ( ...items )
22.1.3.19 Array.prototype.reduce ( callbackfn [ , initialValue ] )
22.1.3.20 Array.prototype.reduceRight ( callbackfn [ , initialValue ] )
22.1.3.21 Array.prototype.reverse ( )
22.1.3.22 Array.prototype.shift ( )
Array.prototype.slice (start, end)
Array.prototype.some (callbackfn[, thisArg])
Array.prototype.sort (comparefn)
 Array.prototype.slice (start, end)
Array.prototype.some (callbackfn[, thisArg])
Array.prototype.sort (comparefn)
 RS: SortCompare(x, y)
Array.prototype.splice (start, deleteCount,...items)
Array.prototype.toLocaleString ([reserved1[, reserved2]])
Array.prototype.toString ()
Array.prototype.unshift (...items)
Array.prototype.values ()
Array.prototype[ @@iterator ] ()
Array.prototype[ @@unscopables ]

Properties of Array Instances

length

Array Iterator Objects

CreateArrayIterator (array, kind)
The %ArrayIteratorPrototype% Object
 %ArrayIteratorPrototype%.next()
 %ArrayIteratorPrototype%[ @@toStringTag ]
Properties of Array Iterator Instances

TypedArray Objects

The %TypedArray% Intrinsic Object
 %TypedArray%()

Properties of the %TypedArray% Intrinsic Object
 %TypedArray%.from (source[, mapfn[, thisArg]])
 RS: IterableToList (items, method)
 %TypedArray%.of (...items)
 %TypedArray%.prototype
 %TypedArray%[ @@species ]

Properties of the %TypedArrayPrototype% Object
 %TypedArray%.prototype.buffer
 %TypedArray%.prototype.byteLength
 %TypedArray%.prototype.byteOffset
 %TypedArray%.prototype.constructor
 %TypedArray%.prototype.copyWithin (target, start[, end])
 RS: ValidateTypedArray (O)
 %TypedArray%.prototype.entries ()
 %TypedArray%.prototype.every (callbackfn[, thisArg])
 %TypedArray%.prototype.fill (value[, start[, end]])
 %TypedArray%.prototype.filter (callbackfn[, thisArg])
 %TypedArray%.prototype.find (predicate[, thisArg])
22.2.3.11 %TypedArray%.prototype.findIndex ( \*predicate [ , thisArg ] \*)
22.2.3.12 %TypedArray%.prototype.forEach ( \*callbackfn [ , thisArg ] \*)
22.2.3.13 %TypedArray%.prototype.includes ( \*searchElement [ , fromIndex ] \*)
22.2.3.14 %TypedArray%.prototype.indexOf ( \*searchElement [ , fromIndex ] \*)
22.2.3.15 %TypedArray%.prototype.join ( \*separator \*)
22.2.3.16 %TypedArray%.prototype.keys ( \*
22.2.3.17 %TypedArray%.prototype.lastIndexOf ( \*searchElement [ , fromIndex ] \*)
22.2.3.18 get %TypedArray%.prototype.length
22.2.3.19 %TypedArray%.prototype.map ( \*callbackfn [ , thisArg ] \*)
22.2.3.20 %TypedArray%.prototype.reduce ( \*callbackfn [ , initialValue ] \*)
22.2.3.21 %TypedArray%.prototype.reduceRight ( \*callbackfn [ , initialValue ] \*)
22.2.3.22 %TypedArray%.prototype.reverse ( \*
22.2.3.23 %TypedArray%.prototype.set ( \*overloaded [ , offset ] \*)
  22.2.3.23.1 %TypedArray%.prototype.set ( \*array [ , offset ] \*)
  22.2.3.23.2 %TypedArray%.prototype.set ( \*typedArray [ , offset ] \*)
22.2.3.24 %TypedArray%.prototype.slice ( \*start, end \*)
22.2.3.25 %TypedArray%.prototype.some ( \*callbackfn [ , thisArg ] \*)
22.2.3.26 %TypedArray%.prototype.sort ( \*comparefn \*)
22.2.3.27 %TypedArray%.prototype.subarray ( \*begin, end \*)
22.2.3.28 %TypedArray%.prototype.toLocaleString ( [ \*reserved1 [ , reserved2 ] \*] \*)
22.2.3.29 %TypedArray%.prototype.toString ( \*
22.2.3.30 %TypedArray%.prototype.values ( \*
22.2.3.31 %TypedArray%.prototype [ @@iterator ] ( \*
22.2.3.32 get %TypedArray%.prototype [ @@toStringTag ]

22.2.4 The **TypedArray** Constructors
  22.2.4.1 **TypedArray** ( \*
  22.2.4.2 **TypedArray** ( \*length \*)
    22.2.4.2.1 RS: AllocateTypedArray ( \*constructorName, newTarget, defaultProto [ , length ] \*)
    22.2.4.2.2 RS: AllocateTypedArrayBuffer ( \*O, length \*)
  22.2.4.3 **TypedArray** ( \*typedArray \*)
  22.2.4.4 **TypedArray** ( \*object \*)
  22.2.4.5 **TypedArray** ( \*buffer [ , byteOffset [ , length ] \*] \*)
  22.2.4.6 TypedArrayCreate ( \*constructor, argumentList \*)
  22.2.4.7 TypedArraySpeciesCreate ( \*exemplar, argumentList \*)

22.2.5 Properties of the **TypedArray** Constructors
  22.2.5.1 **TypedArray.prototype.BYTES_PER_ELEMENT**
  22.2.5.2 **TypedArray.prototype**

22.2.6 Properties of **TypedArray** Prototype Objects
  22.2.6.1 **TypedArray.prototype.BYTES_PER_ELEMENT**
  22.2.6.2 **TypedArray.prototype.constructor**
22.2.7 Properties of *TypedArray* Instances

23 Keyed Collection

23.1 Map Objects

23.1.1 The Map Constructor

23.1.1.1 Map ([ *iterable* ])

23.1.2 Properties of the Map Constructor

23.1.2.1 Map.prototype

23.1.2.2 get Map [ @@species ]

23.1.3 Properties of the Map Prototype Object

23.1.3.1 Map.prototype.clear ( )

23.1.3.2 Map.prototype.constructor

23.1.3.3 Map.prototype.delete ( *key* )

23.1.3.4 Map.prototype.entries ( )

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

23.1.3.6 Map.prototype.get ( *key* )

23.1.3.7 Map.prototype.has ( *key* )

23.1.3.8 Map.prototype.keys ( )

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

23.1.3.10 get Map.prototype.size

23.1.3.11 Map.prototype.values ( )

23.1.3.12 Map.prototype[ @@iterator ] ( )

23.1.3.13 Map.prototype[ @@toStringTag ]

23.1.4 Properties of Map Instances

23.1.5 Map Iterator Objects

23.1.5.1 CreateMapIterator ( *map*, *kind* )

23.1.5.2 The %MapIteratorPrototype% Object

23.1.5.2.1 %MapIteratorPrototype%.next ( )

23.1.5.2.2 %MapIteratorPrototype% [ @@toStringTag ]

23.1.5.3 Properties of Map Iterator Instances

23.2 Set Objects

23.2.1 The Set Constructor

23.2.1.1 Set ([ *iterable* ])

23.2.2 Properties of the Set Constructor

23.2.2.1 Set.prototype

23.2.2.2 get Set [ @@species ]

23.2.3 Properties of the Set Prototype Object

23.2.3.1 Set.prototype.add ( *value* )

23.2.3.2 Set.prototype.clear ( )

23.2.3.3 Set.prototype.constructor

23.2.3.4 Set.prototype.delete ( *value* )
23.2.3.5 Set.prototype.entries ()
23.2.3.6 Set.prototype.forEach (callbackfn [ , thisArg ])
23.2.3.7 Set.prototype.has (value)
23.2.3.8 Set.prototype.keys ()
23.2.3.9 get Set.prototype.size
23.2.3.10 Set.prototype.values ()
23.2.3.11 Set.prototype [ @@iterator ] ()
23.2.3.12 Set.prototype [ @@toStringTag ]

23.2.4 Properties of Set Instances

23.2.5 Set Iterator Objects

23.2.5.1 CreateSetIterator (set, kind)
23.2.5.2 The %SetIteratorPrototype% Object
  23.2.5.2.1 %SetIteratorPrototype%.next ()
  23.2.5.2.2 %SetIteratorPrototype% [ @@toStringTag ]

23.2.5.3 Properties of Set Iterator Instances

23.3 WeakMap Objects

23.3.1 The WeakMap Constructor
  23.3.1.1 WeakMap ([ iterable ])

23.3.2 Properties of the WeakMap Constructor
  23.3.2.1 WeakMap.prototype

23.3.3 Properties of the WeakMap Prototype Object

23.3.3.1 WeakMap.prototype.constructor
  23.3.3.2 WeakMap.prototype.delete (key)
  23.3.3.3 WeakMap.prototype.get (key)
  23.3.3.4 WeakMap.prototype.has (key)
  23.3.3.5 WeakMap.prototype.set (key, value)
  23.3.3.6 WeakMap.prototype [ @@toStringTag ]

23.3.4 Properties of WeakMap Instances

23.4 WeakSet Objects

23.4.1 The WeakSet Constructor
  23.4.1.1 WeakSet ([ iterable ])

23.4.2 Properties of the WeakSet Constructor
  23.4.2.1 WeakSet.prototype

23.4.3 Properties of the WeakSet Prototype Object

23.4.3.1 WeakSet.prototype.add (value)
  23.4.3.2 WeakSet.prototype.constructor
  23.4.3.3 WeakSet.prototype.delete (value)
  23.4.3.4 WeakSet.prototype.has (value)
  23.4.3.5 WeakSet.prototype [ @@toStringTag ]

23.4.4 Properties of WeakSet Instances
24 Structured Data

24.1 ArrayBuffer Objects

24.1.1 Abstract Operations For ArrayBuffer Objects

24.1.1.1 AllocateArrayBuffer ( constructor, byteLength )

24.1.1.2 IsDetachedBuffer ( arrayBuffer )

24.1.1.3 DetachArrayBuffer ( arrayBuffer )

24.1.1.4 CloneArrayBuffer ( srcBuffer, srcByteOffset, srcLength , cloneConstructor )

24.1.1.5 RawBytesToNumber ( type, rawBytes, isLittleEndian )

24.1.1.6 GetValueFromBuffer ( arrayBuffer, byteIndex, type, isTypedArray, order , isLittleEndian )

24.1.1.7 NumberToRawBytes ( type, value, isLittleEndian )

24.1.1.8 SetValueInBuffer ( arrayBuffer, byteIndex, type, value, isTypedArray, order , isLittleEndian )

24.1.1.9 GetModifySetValueInBuffer( arrayBuffer, byteIndex, type, value, op , isLittleEndian )

24.1.2 The ArrayBuffer Constructor

24.1.2.1 ArrayBuffer ( length )

24.1.3 Properties of the ArrayBuffer Constructor

24.1.3.1 ArrayBuffer.isView ( arg )

24.1.3.2 ArrayBuffer.prototype

24.1.3.3 get ArrayBuffer [ @@species ]

24.1.4 Properties of the ArrayBuffer Prototype Object

24.1.4.1 get ArrayBuffer.prototype.byteLength

24.1.4.2 ArrayBuffer.prototype.constructor

24.1.4.3 ArrayBuffer.prototype.slice ( start, end )

24.1.4.4 ArrayBuffer.prototype [ @@toStringTag ]

24.1.5 Properties of the ArrayBuffer Instances

24.2 SharedArrayBuffer Objects

24.2.1 Abstract Operations for SharedArrayBuffer Objects

24.2.1.1 AllocateSharedArrayBuffer ( constructor, byteLength )

24.2.1.2 IsSharedArrayBuffer ( obj )

24.2.2 The SharedArrayBuffer Constructor

24.2.2.1 SharedArrayBuffer ( length )

24.2.3 Properties of the SharedArrayBuffer Constructor

24.2.3.1 SharedArrayBuffer.prototype

24.2.3.2 get SharedArrayBuffer [ @@species ]

24.2.4 Properties of the SharedArrayBuffer Prototype Object

24.2.4.1 get SharedArrayBuffer.prototype.byteLength

24.2.4.2 SharedArrayBuffer.prototype.constructor

24.2.4.3 SharedArrayBuffer.prototype.slice ( start, end )

24.2.4.4 SharedArrayBuffer.prototype [ @@toStringTag ]

24.2.5 Properties of the SharedArrayBuffer Instances
24.3 DataView Objects

24.3.1 Abstract Operations For DataView Objects

24.3.1.1 GetViewValue ( view, requestIndex, isLittleEndian, type )

24.3.1.2 SetViewValue ( view, requestIndex, isLittleEndian, type, value )

24.3.2 The DataView Constructor

24.3.2.1 DataView ( buffer [, byteOffset [, byteLength ] ] )

24.3.3 Properties of the DataView Constructor

24.3.3.1 DataView.prototype

24.3.4 Properties of the DataView Prototype Object

24.3.4.1 get DataView.prototype.buffer

24.3.4.2 get DataView.prototype.byteLength

24.3.4.3 get DataView.prototype.byteOffset

24.3.4.4 DataView.prototype.constructor

24.3.4.5 DataView.prototype.getFloat32 ( byteOffset [, littleEndian ] )

24.3.4.6 DataView.prototype.getFloat64 ( byteOffset [, littleEndian ] )

24.3.4.7 DataView.prototype.getInt8 ( byteOffset )

24.3.4.8 DataView.prototype.getInt16 ( byteOffset [, littleEndian ] )

24.3.4.9 DataView.prototype.getInt32 ( byteOffset [, littleEndian ] )

24.3.4.10 DataView.prototype.getUint8 ( byteOffset )

24.3.4.11 DataView.prototype.getUint16 ( byteOffset [, littleEndian ] )

24.3.4.12 DataView.prototype.getUint32 ( byteOffset [, littleEndian ] )

24.3.4.13 DataView.prototype.setFloat32 ( byteOffset, value [, littleEndian ] )

24.3.4.14 DataView.prototype.setFloat64 ( byteOffset, value [, littleEndian ] )

24.3.4.15 DataView.prototype.setInt8 ( byteOffset, value )

24.3.4.16 DataView.prototype.setInt16 ( byteOffset, value [, littleEndian ] )

24.3.4.17 DataView.prototype.setInt32 ( byteOffset, value [, littleEndian ] )

24.3.4.18 DataView.prototype.setUint8 ( byteOffset, value )

24.3.4.19 DataView.prototype.setUint16 ( byteOffset, value [, littleEndian ] )

24.3.4.20 DataView.prototype.setUint32 ( byteOffset, value [, littleEndian ] )

24.3.4.21 DataView.prototype [ @@toStringTag ]

24.3.5 Properties of DataView Instances

24.4 The Atomics Object

24.4.1 Abstract Operations for Atomics

24.4.1.1 ValidateSharedIntegerTypedArray( typedArray [, onlyInt32 ] )

24.4.1.2 ValidateAtomicAccess( typedArray, requestIndex )

24.4.1.3 GetWaiterList( block, i )

24.4.1.4 EnterCriticalSection( WL )

24.4.1.5 LeaveCriticalSection( WL )

24.4.1.6 AddWaiter( WL, W )

24.4.1.7 RemoveWaiter( WL, W )
24.4.1.8 RemoveWaiters( WL, c )
24.4.1.9 Suspend( WL, W, timeout )
24.4.1.10 WakeWaiter( WL, W )
24.4.1.11 AtomicReadModifyWrite( typedArray, index, value, op )
24.4.1.12 AtomicLoad( typedArray, index )
24.4.2 Atomics.add( typedArray, index, value )
24.4.3 Atomics.and( typedArray, index, value )
24.4.4 Atomics.compareExchange( typedArray, index, expectedValue, replacementValue )
24.4.5 Atomics.exchange( typedArray, index, value )
24.4.6 Atomics.isLockFree( size )
24.4.7 Atomics.load( typedArray, index )
24.4.8 Atomics.or( typedArray, index, value )
24.4.9 Atomics.store( typedArray, index, value )
24.4.10 Atomics.sub( typedArray, index, value )
24.4.11 Atomics.wait( typedArray, index, value, timeout )
24.4.12 Atomics.wake( typedArray, index, count )
24.4.13 Atomics.xor( typedArray, index, value )
24.4.14 Atomics [ @@toStringTag ]

24.5 The JSON Object
24.5.1 JSON.parse( text [, reviver ] )
   24.5.1.1 RS: InternalizeJSONProperty( holder, name )
24.5.2 JSON.stringify( value [, replacer [, space ] ] )
   24.5.2.1 RS: SerializeJSONProperty( key, holder )
   24.5.2.2 RS: QuoteJSONString( value )
   24.5.2.3 RS: SerializeJSONObject( value )
   24.5.2.4 RS: SerializeJSONArray( value )
24.5.3 JSON [ @@toStringTag ]

25 Control Abstraction Objects
25.1 Iteration
   25.1.1 Common Iteration Interfaces
      25.1.1.1 The Iterable Interface
      25.1.1.2 The Iterator Interface
      25.1.1.3 The IteratorResult Interface
   25.1.2 The %IteratorPrototype% Object
      25.1.2.1 %IteratorPrototype% [ @@iterator ] [ ]
25.2 GeneratorFunction Objects
   25.2.1 The GeneratorFunction Constructor
      25.2.1.1 GeneratorFunction( p1, p2, ..., pn, body )
   25.2.2 Properties of the GeneratorFunction Constructor
      25.2.2.1 GeneratorFunction.length
25.2.2.2 GeneratorFunction.prototype
25.2.3 Properties of the GeneratorFunction Prototype Object
   25.2.3.1 GeneratorFunction.prototype.constructor
   25.2.3.2 GeneratorFunction.prototype.prototype
   25.2.3.3 GeneratorFunction.prototype [ @@toStringTag ]
25.2.4 GeneratorFunction Instances
   25.2.4.1 length
   25.2.4.2 name
   25.2.4.3 prototype
25.3 Generator Objects
   25.3.1 Properties of Generator Prototype
      25.3.1.1 Generator.prototype.constructor
      25.3.1.2 Generator.prototype.next ( value )
      25.3.1.3 Generator.prototype.return ( value )
      25.3.1.4 Generator.prototype.throw ( exception )
      25.3.1.5 Generator.prototype [ @@toStringTag ]
   25.3.2 Properties of Generator Instances
   25.3.3 Generator Abstract Operations
      25.3.3.1 GeneratorStart ( generator, generatorBody )
      25.3.3.2 GeneratorValidate ( generator )
      25.3.3.3 GeneratorResume ( generator, value )
      25.3.3.4 GeneratorResumeAbrupt ( generator, abruptCompletion )
      25.3.3.5 GeneratorYield ( iterNextObj )
25.4 Promise Objects
   25.4.1 Promise Abstract Operations
      25.4.1.1 PromiseCapability Records
         25.4.1.1.1 IfAbruptRejectPromise ( value, capability )
      25.4.1.2 PromiseReaction Records
      25.4.1.3 CreateResolvingFunctions ( promise )
         25.4.1.3.1 Promise Reject Functions
         25.4.1.3.2 Promise Resolve Functions
      25.4.1.4 FulfillPromise ( promise, value )
      25.4.1.5 NewPromiseCapability ( C )
         25.4.1.5.1 GetCapabilitiesExecutor Functions
      25.4.1.6 IsPromise ( x )
      25.4.1.7 RejectPromise ( promise, reason )
      25.4.1.8 TriggerPromiseReactions ( reactions, argument )
      25.4.1.9 HostPromiseRejectionTracker ( promise, operation )
   25.4.2 Promise Jobs
      25.4.2.1 PromiseReactionJob ( reaction, argument )
25.4.2.2 PromiseResolveThenableJob (promiseToResolve, thenable, then)
25.4.3 The Promise Constructor
  25.4.3.1 Promise (executor)
25.4.4 Properties of the Promise Constructor
  25.4.4.1 Promise.all (iterable)
    25.4.4.1.1 RS: PerformPromiseAll (iteratorRecord, constructor, resultCapability)
    25.4.4.1.2 Promise.all Resolve Element Functions
  25.4.4.2 Promise.prototype
  25.4.4.3 Promise.race (iterable)
    25.4.4.3.1 RS: PerformPromiseRace (iteratorRecord, constructor, resultCapability)
  25.4.4.4 Promise.reject (r)
  25.4.4.5 Promise.resolve (x)
  25.4.4.6 get Promise [@@species]
25.4.5 Properties of the Promise Prototype Object
  25.4.5.1 Promise.prototype.catch (onRejected)
  25.4.5.2 Promise.prototype.constructor
  25.4.5.3 Promise.prototype.then (onFulfilled, onRejected)
    25.4.5.3.1 PerformPromiseThen (promise, onFulfilled, onRejected, resultCapability)
  25.4.5.4 Promise.prototype [@@toStringTag]
25.4.6 Properties of Promise Instances
25.5 AsyncFunction Objects
  25.5.1 The AsyncFunction Constructor
    25.5.1.1 AsyncFunction (p1, p2, …, pn, body)
  25.5.2 Properties of the AsyncFunction Constructor
    25.5.2.1 AsyncFunction.length
    25.5.2.2 AsyncFunction.prototype
  25.5.3 Properties of the AsyncFunction Prototype Object
    25.5.3.1 AsyncFunction.prototype.constructor
    25.5.3.2 AsyncFunction.prototype [@@toStringTag]
  25.5.4 AsyncFunction Instances
    25.5.4.1 length
    25.5.4.2 name
  25.5.5 Async Functions Abstract Operations
    25.5.5.1 AsyncFunctionCreate (kind, parameters, body, Scope, Strict)
    25.5.5.2 AsyncFunctionStart (promiseCapability, asyncFunctionBody)
    25.5.5.3 AsyncFunctionAwait (value)
    25.5.5.4 AsyncFunction Awaited Fulfilled
    25.5.5.5 AsyncFunction Awaited Rejected
26 Reflection
  26.1 The Reflect Object
26.1.1 Reflect.apply (target, thisArgument, argumentsList)
26.1.2 Reflect.construct (target, argumentsList[, newTarget])
26.1.3 Reflect.defineProperty (target, propertyKey, attributes)
26.1.4 Reflect.deleteProperty (target, propertyKey)
26.1.5 Reflect.get (target, propertyKey[, receiver])
26.1.6 Reflect.getOwnPropertyDescriptor (target, propertyKey)
26.1.7 Reflect.getPrototypeOf (target)
26.1.8 Reflect.has (target, propertyKey)
26.1.9 Reflect.isExtensible (target)
26.1.10 Reflect.ownKeys (target)
26.1.11 Reflect.preventExtensions (target)
26.1.12 Reflect.set (target, propertyKey, V[, receiver])
26.1.13 Reflect.setPrototypeOf (target, proto)

26.2 Proxy Objects
26.2.1 The Proxy Constructor
   26.2.1.1 Proxy (target, handler)
26.2.2 Properties of the Proxy Constructor
   26.2.2.1 Proxy.revocable (target, handler)
      26.2.2.1.1 Proxy Revocation Functions

26.3 Module Namespace Objects
   26.3.1 @@toStringTag

27 Memory Model
27.1 Memory Model Fundamentals
27.2 Agent Events Records
27.3 Chosen Value Records
27.4 Candidate Executions

27.5 Abstract Operations for the Memory Model
   27.5.1 EventSet (execution)
   27.5.2 SharedDataBlockEventSet (execution)
   27.5.3 HostEventSet (execution)
   27.5.4 ComposeWriteEventBytes (execution, byteIndex, Ws)
   27.5.5 ValueOfReadEvent (execution, R)

27.6 Relations of Candidate Executions
   27.6.1 agent-order
   27.6.2 reads-bytes-from
   27.6.3 reads-from
   27.6.4 host-synchronizes-with
   27.6.5 synchronizes-with
   27.6.6 happens-before

27.7 Properties of Valid Executions
27.7.1 Valid Chosen Reads
27.7.2 Coherent Reads
27.7.3 Tear Free Reads
27.7.4 Sequentially Consistent Atomics
27.7.5 Valid Executions
27.8 Races
27.9 Data Races
27.10 Data Race Freedom
27.11 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 Pattern Semantics
         B.1.4.1.1 RS: 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 RS: 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.2.4 Additional Properties of the Date.prototype Object

B.2.4.1 Date.prototype.getYear ()
B.2.4.2 Date.prototype.setYear (year)
B.2.4.3 Date.prototype.toGMTString ()

B.2.5 Additional Properties of the RegExp.prototype Object

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

B.3 Other Additional Features

B.3.1 __proto__ Property Names in Object Initializers
B.3.2 Labelled Function Declarations
B.3.3 Block-Level Function Declarations Web Legacy Compatibility Semantics
  B.3.3.1 Changes to FunctionDeclarationInstantiation
  B.3.3.2 Changes to GlobalDeclarationInstantiation
  B.3.3.3 Changes to EvalDeclarationInstantiation
  B.3.3.4 Changes to Block SS: Early Errors
  B.3.3.5 Changes to switch Statement SS: Early Errors
  B.3.3.6 Changes to BlockDeclarationInstantiation
B.3.4 FunctionDeclarations in IfStatement Statement Clauses
B.3.5 VariableStatements in Catch Blocks
B.3.6 Initializers in ForIn Statement Heads

C The Strict Mode of ECMAScript
D Corrections and Clarifications in ECMAScript 2015 with Possible Compatibility Impact
E Additions and Changes That Introduce Incompatibilities with Prior Editions
F Bibliography
G Copyright & Software License
Introduction

This Ecma Standard defines the ECMAScript 2017 Language. It is the eighth 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.

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

Focused development of the sixth edition started in 2009, as the fifth edition was being prepared for publication. However, this was preceded by significant experimentation and language enhancement design efforts dating to the publication of the third edition in 1999. In a very real sense, the completion of the sixth edition is the culmination of a fifteen year effort. The goals for this addition 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`.

This specification introduces 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. This specification also includes new static methods on `Object`: `Object.values`, `Object.entries`, and `Object.getOwnPropertyDescriptors`.

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, 6th Edition Project Editor
1 Scope

This Standard defines the ECMAScript 2017 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 that supports programs that need to adapt to the linguistic and cultural conventions used by different human languages and countries must implement the interface defined by the most recent edition of ECMA-402 that is compatible with this specification.

A conforming implementation of ECMAScript may provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript may provide properties not described in this specification, and values for those properties, for objects that are described in this specification.

A conforming implementation of ECMAScript may support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript may support program syntax that makes use of the “future reserved words” listed in subclause 11.6.2.2 of this specification.

A conforming implementation of ECMAScript must not implement any extension that is listed as a Forbidden Extension in subclause 16.2.

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.
http://www.ecma-international.org/publications/standards/Ecma-402.htm

ECMA-404, The JSON Data Interchange Format.

4 Overview

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

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

ECMAScript was originally designed to be used as a scripting language, but has become widely used as a general-purpose programming language. A scripting language is a programming language that is used to manipulate, customize, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scripting language is a mechanism for exposing that functionality to program control. In this way, the existing system is said to provide a host environment of objects and facilities, which completes the capabilities of the scripting language. A scripting language is intended for use by both professional and non-professional programmers.

ECMAScript was originally designed to be a Web scripting language, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript is now used to provide core scripting capabilities for a variety of host environments. Therefore the core language is specified in this document apart from any particular host environment.

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

Some of the facilities of ECMAScript are similar to those used in other programming languages; in particular C, Java™, Self, and Scheme as described in:


4.1 Web Scripting

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. Further, the host environment provides a means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. Scripting code appears within the HTML and the displayed page is a combination of user interface elements and fixed and computed text and images. The scripting code is reactive to user interaction, and there is no need for a main program.

A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data. By using browser-side and server-side scripting together, it is possible to distribute computation between the client and server while providing a customized user interface for a Web-based application.

Each Web browser and server that supports ECMAScript supplies its own host environment, completing the ECMAScript execution environment.

4.2 ECMAScript Overview

The following is an informal overview of ECMAScript—not all parts of the language are described. This overview is not part of the standard proper.

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. In ECMAScript, an object is a collection of 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, 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.

### 4.2.1 Objects

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

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

*Figure 1: Object/Prototype Relationships*

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

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

**CF** is a constructor (and also an object). Five objects have been created by using `new` expressions: `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**. The constructor, **CF**, has two properties itself, named `P1` and `P2`, which are not visible to **CF**. **CF**, `cf_1, cf_2, cf_3, cf_4, or cf_5`. The property
named $\text{CFP}_1$ in $\text{CF}_p$ is shared by $\text{cf}_1$, $\text{cf}_2$, $\text{cf}_3$, $\text{cf}_4$, and $\text{cf}_5$ (but not by $\text{CF}$), as are any properties found in $\text{CF}_p$’s implicit prototype chain that are not named $q_1$, $q_2$, or $\text{CFP}_1$. Notice that there is no implicit prototype link between $\text{CF}$ and $\text{CF}_p$.

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 $\text{cf}_1$, $\text{cf}_2$, $\text{cf}_3$, $\text{cf}_4$, and $\text{cf}_5$ by assigning a new value to the property in $\text{CF}_p$.

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.

### 4.2.2 The Strict Variant of ECMAScript

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

The strict variant of ECMAScript is commonly referred to as the strict mode of the language. Strict mode selection and use of the strict mode syntax and semantics of ECMAScript is explicitly made at the level of individual ECMAScript source text units. 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.
4.3 Terms and Definitions

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

4.3.1 type

set of data values as defined in clause 6 of this specification

4.3.2 primitive value

member of one of the types Undefined, Null, Boolean, Number, Symbol, or String as defined in clause 6

NOTE A primitive value is a datum that is represented directly at the lowest level of the language implementation.

4.3.3 object

member of the type Object

NOTE An object is a collection of properties and has a single prototype object. The prototype may be the null value.

4.3.4 constructor

function object that creates and initializes objects

NOTE The value of a constructor's prototype property is a prototype object that is used to implement inheritance and shared properties.

4.3.5 prototype

object that provides shared properties for other objects

NOTE When a constructor creates an object, that object implicitly references the constructor’s prototype property for the purpose of resolving property references. The constructor’s prototype property can be referenced by the program expression constructor.prototype, and properties added to an object’s prototype are shared, through inheritance, by all objects sharing the prototype. Alternatively, a new object may be created with an explicitly specified prototype by using the Object.create built-in function.
4.3.6 ordinary object
object that has the default behaviour for the essential internal methods that must be supported by all objects

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

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

4.3.8 standard object
object whose semantics are defined by this specification

4.3.9 built-in object
object specified and supplied by an ECMAScript implementation

**NOTE** Standard built-in objects are defined in this specification. An ECMAScript implementation may specify and supply additional kinds of built-in objects. A *built-in constructor* is a built-in object that is also a constructor.

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

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

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

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

4.3.14 Boolean value
member of the Boolean type

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

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

4.3.16 Boolean object
member of the Object type that is an instance of the standard built-in Boolean constructor

NOTE A Boolean object is created by using the Boolean constructor in a new expression, supplying a Boolean value as an argument. The resulting object has an internal slot whose value is the Boolean value. A Boolean object can be coerced to a Boolean value.

4.3.17 String value
primitive value that is a finite ordered sequence of zero or more 16-bit unsigned integer 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.

4.3.18 String type
set of all possible String values

4.3.19 String object
member of the Object type that is an instance of the standard built-in String constructor

NOTE A String object is created by using the String constructor in a new expression, supplying a String value as an argument. The resulting object has an internal slot whose value is the String value. A String object can be coerced to a String value by calling the String constructor as a function (21.1.1.1).

4.3.20 Number value
primitive value corresponding to a double-precision 64-bit binary format IEEE 754-2008 value
A Number value is a member of the Number type and is a direct representation of a number.

### 4.3.21 Number type

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

### 4.3.22 Number object

member of the Object type that is an instance of the standard built-in `Number` constructor

A Number object is created by using the `Number` constructor in a `new` expression, supplying a number value as an argument. The resulting object has an internal slot whose value is the number value. A Number object can be coerced to a number value by calling the `Number` constructor as a function (20.1.1).

### 4.3.23 Infinity

number value that is the positive infinite number value

### 4.3.24 NaN

number value that is an IEEE 754-2008 “Not-a-Number” value

### 4.3.25 Symbol value

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

### 4.3.26 Symbol type

set of all possible Symbol values

### 4.3.27 Symbol object

member of the Object type that is an instance of the standard built-in `Symbol` constructor

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

**NOTE**  
In addition to its properties, a function contains executable code and state that determine how it behaves when invoked. A function's code may or may not be written in ECMAScript.

### 4.3.29 built-in function

A built-in object that is a function

**NOTE**  
Examples of built-in functions include `parseInt` and `Math.exp`. An implementation may provide implementation-dependent built-in functions that are not described in this specification.

### 4.3.30 property

A 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.

### 4.3.31 method

A function that is the value of a property

**NOTE**  
When a function is called as a method of an object, the object is passed to the function as its `this` value.

### 4.3.32 built-in method

A method that is a built-in function

**NOTE**  
Standard built-in methods are defined in this specification, and an ECMAScript implementation may specify and provide other additional built-in methods.

### 4.3.33 attribute

An internal value that defines some characteristic of a property

### 4.3.34 own property
property that is directly contained by its object

4.3.35 inherited property

property of an object that is not an own property but is a property (either own or inherited) of the object’s prototype

4.4 Organization of This Specification

The remainder of this specification is organized as follows:

Clause 5 defines the notational conventions used throughout the specification.

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

Clauses 10-16 define the actual ECMAScript programming language including its syntactic encoding and the execution semantics of all language features.

Clauses 17-26 define the ECMAScript standard library. They include the definitions of all of the standard objects that are available for use by ECMAScript programs as they execute.

5 Notational Conventions

5.1 Syntactic and Lexical Grammars

5.1.1 Context-Free Grammars

A context-free grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of zero or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

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

Starting from a sentence consisting of a single distinguished nonterminal, called the goal symbol, a given context-free grammar specifies a language, namely, the (perhaps infinite) set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.
5.1.2 The Lexical and RegExp Grammars

A *lexical grammar* for ECMAScript is given in clause 11. This grammar has as its terminal symbols Unicode code points that conform to the rules for *SourceCharacter* defined in 10.1. It defines a set of productions, starting from the goal 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 (11.9). Simple white space and single-line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that is, a comment of the form /*...*/ regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a *MultiLineComment* contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

A *RegExp grammar* for ECMAScript is given in 21.2.1. This grammar also has as its terminal symbols the code points as defined by *SourceCharacter*. It defines a set of productions, starting from the goal 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 7.1.3.1.

Productions of the numeric string grammar are distinguished by having three colons “:::” as punctuation.

5.1.4 The Syntactic Grammar

The *syntactic grammar* for ECMAScript is given in clauses 11, 12, 13, 14, and 15. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (5.1.2). It defines a set of productions, starting from 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.

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

The syntactic grammar as presented in clauses 12, 13, 14 and 15 is 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. In certain contexts, when explicitly specified, the input elements corresponding to such a production are parsed again using a goal symbol of a supplemental grammar. The input stream is syntactically in error if the tokens in the stream of input elements parsed by a cover grammar cannot be parsed as a single instance of the corresponding supplemental goal symbol, with no tokens left over.

### 5.1.5 Grammar Notation

Terminal symbols of the lexical, RegExp, and numeric string grammars are shown in fixed width font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a script exactly as written. All terminal symbol code points specified in this way are to be understood as the appropriate Unicode code points from the Basic Latin range, as opposed to any similar-looking code points from other Unicode ranges.

Nonterminal symbols are shown in italic type. The definition of a nonterminal (also called a “production”) is introduced by the name of the nonterminal being defined followed by one or more colons. (The number
of colons indicates to which grammar the production belongs.) One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

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

states that the nonterminal \text{WhileStatement} represents the token \textbf{while}, followed by a left parenthesis token, followed by an \textit{Expression}, followed by a right parenthesis token, followed by a \textit{Statement}. The occurrences of \textit{Expression} and \textit{Statement} are themselves nonterminals. As another example, the syntactic definition:

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

states that an \textit{ArgumentList} may represent either a single \textit{AssignmentExpression} or an \textit{ArgumentList}, followed by a comma, followed by an \textit{AssignmentExpression}. This definition of \textit{ArgumentList} is recursive, that is, it is defined in terms of itself. The result is that an \textit{ArgumentList} may contain any positive number of arguments, separated by commas, where each argument expression is an \textit{AssignmentExpression}. Such recursive definitions of nonterminals are common.

The subscripted suffix “\texttt{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:

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

is a convenient abbreviation for:

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

and that:

\[
\text{IterationStatement} : \\
\quad \text{for} ( \text{LexicalDeclaration} \text{Expression}_{\texttt{opt}} ; \text{Expression}_{\texttt{opt}} ) \text{Statement}
\]

is a convenient abbreviation for:

\[
\text{IterationStatement} : \\
\quad \text{for} ( \text{LexicalDeclaration} ; \text{Expression}_{\texttt{opt}} ) \text{Statement} \\
\quad \text{for} ( \text{LexicalDeclaration} \text{Expression} ; \text{Expression}_{\texttt{opt}} ) \text{Statement}
\]
which in turn is an abbreviation for:

\[
\text{IterationStatement} : \\
\quad \text{for ( LexicalDeclaration ; ) Statement} \\
\quad \text{for ( LexicalDeclaration ; Expression ) Statement} \\
\quad \text{for ( LexicalDeclaration Expression ; ) Statement} \\
\quad \text{for ( LexicalDeclaration Expression ; Expression ) Statement}
\]

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

A production may be parameterized by a subscripted annotation of the form “[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:

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

is a convenient abbreviation for:

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

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

and that:

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

is an abbreviation for:

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

\[
\text{StatementList_Return} : \\
\quad \text{ReturnStatement}
\]
Multiple parameters produce a combinatory number of productions, not all of which are necessarily referenced in a complete grammar.

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

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

is equivalent to saying:

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

and:

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

is equivalent to:

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

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

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

is an abbreviation for:

\[
\text{VariableDeclaration} : \\
\]
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:

\[
StatementList[Return] :
\]

\[
[\text{+Return}] \ ReturnStatement
\]

ExpressionStatement

is an abbreviation for:

\[
StatementList :
\]

ExpressionStatement

\[
StatementList_Return :
\]

ReturnStatement

ExpressionStatement

and that:

\[
StatementList[Return] :
\]

\[
[\text{~Return}] \ ReturnStatement
\]

ExpressionStatement

is an abbreviation for:

\[
StatementList :
\]
ReturnStatement
ExpressionStatement

StatementList_Return :
ExpressionStatement

When the words “one of” follow the colon(s) in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for ECMAScript contains the production:

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

which is merely a convenient abbreviation for:

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

If the phrase “[empty]” appears as the right-hand side of a production, it indicates that the production's right-hand side contains no terminals or nonterminals.

If the phrase “[lookahead ∉ set]” appears in the right-hand side of a production, it indicates that the production may not be used if the immediately following input token sequence is a member of the given set. The set can be written as a comma separated list of one or two element terminal sequences enclosed in curly brackets. For convenience, the set can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand. If the set consists of a single terminal the phrase “[lookahead ≠ terminal]” may be used.

For example, given the definitions:

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

DecimalDigits ::
DecimalDigit
DecimalDigits DecimalDigit
the definition:

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

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.

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:

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

indicates that the production may not be used if a LineTerminator occurs in the script between the \text{throw} token and the \text{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 “\textbf{but not}” and then indicating the expansions to be excluded. For example, the production:

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

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

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

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

**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, in which case the names and usage of the parameters must be provided as part of the algorithm’s definition. In order to facilitate their use in multiple parts of this specification, some algorithms, called abstract operations, are named and written in parameterized functional form so that they may be referenced by name from within other algorithms. Abstract operations are typically referenced using a functional application style such as operationName(arg1, arg2). Some abstract operations are treated as polymorphically dispatched methods of class-like specification abstractions. Such method-like abstract operations are typically referenced using a method application style such as someValue.operationName(arg1, arg2).

Calls to abstract operations return Completion Records. Abstract operations referenced using the functional application style and the method application style that are prefixed by ? indicate that ReturnIfAbrupt should be applied to the resulting Completion Record. For example, ?operationName() is equivalent to ReturnIfAbrupt(operationName()). Similarly, ?someValue.operationName() is equivalent to ReturnIfAbrupt(someValue.operationName()).

The prefix ! is used to indicate that an abstract 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, “Let val be ! operationName()” is equivalent to the following algorithm steps:

1. Let val be operationName().
2. Assert: val is never an abrupt completion.
3. If val is a Completion Record, let val be val.[[Value]].

Algorithms may be associated with productions of one of the ECMAScript grammars. A production that has multiple alternative definitions will typically have a distinct algorithm for each alternative. When an algorithm is associated with a grammar production, it may reference the terminal and nonterminal symbols of the production alternative as if they were parameters of the algorithm. When used in this manner, nonterminal symbols refer to the actual alternative definition that is matched when parsing the source text.

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

Unless explicitly specified otherwise, all chain productions have an implicit definition for every algorithm that might be applied to that production’s left-hand side nonterminal. The implicit definition simply reapplyes the same algorithm name with the same parameters, if any, to the chain production’s sole right-hand side nonterminal and then returns the result. For example, assume there is a production:
but there is no corresponding Evaluation algorithm that is explicitly specified for that production. If in some algorithm there is a statement of the form: “Return the result of evaluating Block” it is implicit that an Evaluation algorithm exists of the form:

**Runtime Semantics: Evaluation**

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

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

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

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

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

A step may specify the iterative application of its substeps.

A step 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 \text{someValue}”. These aliases are reference-like in that both \(x\) and \text{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 \text{someValue}” creates a shallow copy of \text{someValue}.
Mathematical operations such as addition, subtraction, negation, multiplication, division, and the mathematical functions defined later in this clause should always be understood as computing exact mathematical results on mathematical real numbers, which unless otherwise noted do not include infinities and do not include a negative zero that is distinguished from positive zero. Algorithms in this standard that model floating-point arithmetic include explicit steps, where necessary, to handle infinities and signed zero and to perform rounding. If a mathematical operation or function is applied to a floating-point number, it should be understood as being applied to the exact mathematical value represented by that floating-point number; such a floating-point number must be finite, and if it is +0 or -0 then the corresponding mathematical value is simply 0.

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

The mathematical function min(x1, x2, ..., xN) produces the mathematically smallest of x1 through xN. The mathematical function max(x1, x2, ..., xN) produces the mathematically largest of x1 through xN. The domain and range of these mathematical functions include +∞ and -∞.

The notation “x modulo y” (y must be finite and nonzero) computes a value k of the same sign as y (or zero) such that abs(k) < abs(y) and x-k = q × y for some integer q.

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

NOTE floor(x) = x-(x modulo 1).

5.3 Static Semantic Rules

Context-free grammars are not sufficiently powerful to express all the rules that define whether a stream of input elements form a valid ECMAScript Script 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.

Unless otherwise specified every grammar production alternative in this specification implicitly has a definition for a static semantic rule named Contains which takes an argument named symbol whose value is a terminal or nonterminal of the grammar that includes the associated production. The default definition of Contains is:
1. For each child node \textit{child} of this Parse Node, do
   a. If \textit{child} is an instance of \textit{symbol}, return \textit{true}.
   b. If \textit{child} is an instance of a nonterminal, then
      i. Let \textit{contained} be the result of \textit{child} Contains \textit{symbol}.
      ii. If \textit{contained} is \textit{true}, return \textit{true}.
 2. Return \textit{false}.

The above definition is explicitly over-ridden for specific productions.

A special kind of static semantic rule is an \textit{Early Error Rule}. \textit{Early error} rules define \textit{early error} conditions (see clause 16) that are associated with specific grammar productions. Evaluation of most \textit{early error} rules are not explicitly invoked within the algorithms of this specification. A conforming implementation must, prior to the first evaluation of a \textit{Script} or \textit{Module}, validate all of the \textit{early error} rules of the productions used to parse that \textit{Script} or \textit{Module}. If any of the \textit{early error} rules are violated the \textit{Script} or \textit{Module} is invalid and cannot be evaluated.

6 ECMAScript Data Types and Values

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

Within this specification, the notation “\textit{Type(x)}” is used as shorthand for “the type of \textit{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 \textit{ECMAScript language type} corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are \textit{Undefined}, \textit{Null}, \textit{Boolean}, \textit{String}, \textit{Symbol}, \textit{Number}, and \textit{Object}. An \textit{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 \textit{undefined}. Any variable that has not been assigned a value has the value \textit{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 nonnegative integers. The first element (if any) is at index 0, the next element (if any) at index 1, and so on. The length of a String is the number of elements (i.e., 16-bit values) within it. The empty String has length zero and therefore contains no elements.

Where ECMAScript operations interpret String values, each element is interpreted as a single UTF-16 code unit. However, ECMAScript does not place any restrictions or requirements on the sequence of code units in a String value, so they may be ill-formed when interpreted as UTF-16 code unit sequences. Operations that do not interpret String contents treat them as sequences of undifferentiated 16-bit unsigned integers. The function **String.prototype.normalize** (see 21.1.3.12) can be used to explicitly normalize a String value. **String.prototype.localeCompare** (see 21.1.3.10) 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.
```

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

- A code unit in the range 0 to 0xD7FF or in the range 0xE000 to 0xFFFF is interpreted as a code point with the same value.
- A sequence of two code units, where the first code unit $c1$ is in the range 0xD800 to 0xDBFF and the second code unit $c2$ is in the range 0xDC00 to 0xDFFF, is a surrogate pair and is interpreted as a code point with the value $(c1 - 0xD800) \times 0x400 + (c2 - 0xDC00) + 0x10000$. (See 10.1.2)
- A code unit that is in the range 0xD800 to 0xDFFF, but is not part of a surrogate pair, is interpreted as a code point with the same value.
6.1.5 The Symbol Type

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

Each possible Symbol value is unique and immutable.

Each Symbol value immutably holds an associated value called `[[Description]]` that is either `undefined` or a String value.

6.1.5.1 Well-Known Symbols

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

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

<table>
<thead>
<tr>
<th>Specification Name</th>
<th><code>[[Description]]</code></th>
<th>Value and Purpose</th>
</tr>
</thead>
<tbody>
<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 <code>instanceof</code> 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 <code>Array.prototype.concat</code>.</td>
</tr>
<tr>
<td>@@iterator</td>
<td>&quot;Symbol.iterator&quot;</td>
<td>A method that returns the default Iterator for an object. Called by the semantics of the for-of statement.</td>
</tr>
<tr>
<td>@@match</td>
<td>&quot;Symbol.match&quot;</td>
<td>A regular expression method that matches the regular expression against a string. Called by the <code>String.prototype.match</code> 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.</td>
</tr>
</tbody>
</table>

Table 1: Well-known Symbols
<table>
<thead>
<tr>
<th>@@search</th>
<th>&quot;Symbol.search&quot;</th>
<th>A regular expression method that returns the index within a string that matches the regular expression. Called by the <code>String.prototype.replace</code> method.</th>
</tr>
</thead>
<tbody>
<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 <code>String.prototype.split</code> 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 <code>ToPrimitive</code> 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 <code>Object.prototype.toString</code>.</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 <code>with</code> environment bindings of the associated object.</td>
</tr>
</tbody>
</table>

### 6.1.6 The Number Type

The Number type has exactly $18437736874454810627$ (that is, $2^{64} - 2^{53} + 3$) values, representing the double-precision 64-bit format IEEE 754-2008 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the $9007199254740990$ (that is, $2^{53} - 2$) distinct “Not-a-Number” values of the IEEE Standard are represented in ECMAScript as a single special `NaN` value. (Note that the `NaN` value is produced by the program expression `NaN`.) In some implementations, external code might be
able to detect a difference between various Not-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all NaN values are indistinguishable from each other.

**NOTE**
The bit pattern that might be observed in an ArrayBuffer (see 24.1) or a SharedArrayBuffer (see 24.2) 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 +∞ and −∞, respectively. (Note that these two infinite Number values are produced by the program expressions +Infinity (or simply Infinity) and -Infinity.)

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

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

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

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

$$s \times m \times 2^e$$

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

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

$$s \times m \times 2^e$$

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

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

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

In this specification, the phrase “the Number value for $x$” where $x$ represents an exact nonzero real mathematical quantity (which might even be an irrational number such as π) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with -0 removed
and with two additional values added to it that are not representable in the Number type, namely $2^{1024}$ (which is $+1 \times 2^{53} \times 2^{971}$) and $-2^{1024}$ (which is $-1 \times 2^{53} \times 2^{971}$). Choose the member of this set that is closest in value to $x$. If two values of the set are equally close, then the one with an even significand is chosen; for this purpose, the two extra values $2^{1024}$ and $-2^{1024}$ are considered to have even significands. Finally, if $2^{1024}$ was chosen, replace it with $+\infty$; if $-2^{1024}$ was chosen, replace it with $-\infty$; if $+0$ was chosen, replace it with $-0$ if and only if $x$ is less than zero; any other chosen value is used unchanged. The result is the Number value for $x$. (This procedure corresponds exactly to the behaviour of the IEEE 754-2008 “round to nearest, ties to even” mode.)

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.

### 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 7.1.16) and whose numeric value is either $+0$ or a positive integer $\leq 2^{53}-1$. An **array index** is an integer index whose numeric value $i$ is in the range $+0 \leq i < 2^{32}-1$.

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. *Ordinary objects* are the most common form of objects and have the default object semantics. An *exotic object* is any form of object whose property semantics differ in any way from the default semantics.

### 6.1.7.1 Property Attributes

Attributes are used in this specification to define and explain the state of Object properties. A data property associates a key value with the attributes listed in Table 2.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>Any ECMAScript language type</td>
<td>The value retrieved by a get access of the property.</td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>Boolean</td>
<td>If false, attempts by ECMAScript code to change the property's [[Value]] attribute using [[Set]] will not succeed.</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>Boolean</td>
<td>If true, the property will be enumerated by a for-in enumeration (see 13.7.5). Otherwise, the property is said to be non-enumarable.</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>Boolean</td>
<td>If false, attempts to delete the property, change the property to be an accessor property, or change its attributes (other than [[Value]], or changing [[Writable]] to false) will fail.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Get]]</td>
<td>Object</td>
<td>Undefined</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>Object</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
assigned value as its sole argument each time a set access of the property is performed. The effect of a property's [[Set]] internal method may, but is not required to, have an effect on the value returned by subsequent calls to the property's [[Get]] internal method.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[[[Enumerable]]]</td>
<td>Boolean</td>
<td>If <strong>true</strong>, the property is to be enumerated by a for-in enumeration (see 13.7.5). Otherwise, the property is said to be non-enumerable.</td>
</tr>
<tr>
<td>[[[Configurable]]]</td>
<td>Boolean</td>
<td>If <strong>false</strong>, attempts to delete the property, change the property to be a data property, or change its attributes will fail.</td>
</tr>
</tbody>
</table>

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

<table>
<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 \texttt{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 \texttt{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 [[ ]].

\textbf{Table 5} summarizes the essential internal methods used by this specification that are applicable to all objects created or manipulated by ECMAScript code. Every object must have algorithms for all of the essential internal methods. However, all objects do not necessarily use the same algorithms for those methods.

The “Signature” column of \textbf{Table 5} and other similar tables describes the invocation pattern for each internal method. The invocation pattern always includes a parenthesized list of descriptive parameter names. If a parameter name is the same as an ECMAScript type name then the name describes the required type of the parameter value. If an internal method explicitly returns a value, its parameter list is followed by the symbol “→” and the type name of the returned value. The type names used in signatures refer to the types defined in clause 6 augmented by the following additional names. “any” means the value may be any ECMAScript language type. An internal method implicitly returns a Completion Record. In addition to its parameters, an internal method always has access to the object that is the target of the method invocation.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Internal Method} & \textbf{Signature} & \textbf{Description} \\
\hline
[[GetPrototypeOf]] & () → Object | Null & Determine the object that provides inherited properties for this object. A \texttt{null} value indicates that there are no inherited properties. \\
\hline
[[SetPrototypeOf]] & (Object | Null) → Boolean & Associate this object with another object that provides inherited properties. Passing \texttt{null} indicates that there are no inherited properties. Returns \texttt{true} indicating that the operation was completed successfully or \texttt{false} indicating that the operation was not successful. \\
\hline
[[IsExtensible]] & () → Boolean & Determine whether it is permitted to add additional properties to this object. \\
\hline
\end{tabular}
\caption{Essential Internal Methods}
\end{table}
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[PreventExtensions]]</td>
<td>() → Boolean</td>
<td>Control whether new properties may be added to this object. Returns <code>true</code> if the operation was successful or <code>false</code> if the operation was unsuccessful.</td>
</tr>
<tr>
<td>[[GetOwnProperty]]</td>
<td>(propertyKey) → Undefined</td>
<td>Property Descriptor</td>
</tr>
<tr>
<td>[[DefineOwnProperty]]</td>
<td>(propertyKey, PropertyDescriptor) → Boolean</td>
<td>Create or alter the own property, whose key is <code>propertyKey</code>, to have the state described by <code>PropertyDescriptor</code>. Return <code>true</code> if that property was successfully created/updated or <code>false</code> 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 <code>propertyKey</code>.</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>(propertyKey, Receiver) → any</td>
<td>Return the value of the property whose key is <code>propertyKey</code> from this object. If any ECMAScript code must be executed to retrieve the property value, <code>Receiver</code> is used as the <code>this</code> 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 <code>propertyKey</code> to <code>value</code>. If any ECMAScript code must be executed to set the property value, <code>Receiver</code> is used as the <code>this</code> value when evaluating the code. Returns <code>true</code> if the property value was set or <code>false</code> if it could not be set.</td>
</tr>
<tr>
<td>[[Delete]]</td>
<td>(propertyKey) → Boolean</td>
<td>Remove the own property whose key is <code>propertyKey</code> from this object. Return <code>false</code> if the property was not deleted and is still present. Return <code>true</code> if the property was deleted or is not present.</td>
</tr>
<tr>
<td>[[OwnPropertyKeys]]</td>
<td>() → List of propertyKey</td>
<td>Return a <code>List</code> whose elements are all of the own property keys for the object.</td>
</tr>
</tbody>
</table>

Table 6 summarizes additional essential internal methods that are supported by objects that may be called as functions. A `function object` is an object that supports the `[[Call]]` internal method. A `constructor` (also referred to as a `constructor function`) is a function object that supports the `[[Construct]]` internal method.
Table 6: Additional Essential Internal Methods of Function Objects

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Call]]</td>
<td>(any, a List of any) → any</td>
<td>Executes code associated with this object. Invoked via a function call expression. The arguments to the internal method are a <code>this</code> value and a list containing the arguments passed to the function by a call expression. Objects that implement this internal method are callable.</td>
</tr>
<tr>
<td>[[Construct]]</td>
<td>(a List of any, Object) → Object</td>
<td>Creates an object. Invoked via the <code>new</code> or <code>super</code> operators. The first argument to the internal method is a list containing the arguments of the operator. The second argument is the object to which the <code>new</code> 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 9. If any specified use of an internal method of an exotic object is not supported by an implementation, that usage must throw a `TypeError` exception when attempted.

6.1.7.3 Invariants of the Essential Internal Methods

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

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

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

Definitions:

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

[[GetPrototypeOf]] ( )

• The Type of the return value must be either Object or Null.
• If target is non-extensible, and [[GetPrototypeOf]] returns a value v, then any future calls to [[GetPrototypeOf]] should return the *SameValue* as v.

**NOTE 1**
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 Type of the return value must be Boolean.
• If target is non-extensible, [[SetPrototypeOf]] must return false, unless V is the *SameValue* as the target’s observed [[GetPrototypeOf]] value.

[[IsExtensible]] ( )

• The Type of the return value must be Boolean.
• If [[IsExtensible]] returns false, all future calls to [[IsExtensible]] on the target must return false.

[[PreventExtensions]] ( )

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

[[GetOwnProperty]] (P)

• The Type of the return value must be either *Property Descriptor* or Undefined.
• If the Type of the return value is *Property Descriptor*, the return value must be a complete property descriptor (see 6.2.5.6).
• If a property P is described as a data property with Desc.[[Value]] equal to v and Desc.[[Writable]] and Desc.[[Configurable]] are both false, then the *SameValue* must be returned for the Desc.[[Value]] attribute of the property on all future calls to [[GetOwnProperty]] (P).
• If P’s attributes other than [[Writable]] may change over time or if the property might disappear, then P’s [[Configurable]] attribute must be true.
If the [[Writable]] attribute may change from false to true, then the [[Configurable]] attribute must be true.

If the target is non-extensible and P is non-existent, then all future calls to [[GetOwnProperty]] (P) on the target must describe P as non-existent (i.e. [[GetOwnProperty]] (P) must return undefined).

NOTE 2  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 Desc.[[Writable]] and Desc.[[Configurable]] attributes must be true even if no mechanism to change the value is exposed via the other internal methods.

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

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

[[HasProperty]] \((P)\)

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

[[Get]] \((P, \text{Receiver})\)

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

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

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

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

[[OwnPropertyKeys]] ()

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

[[Construct]] ()

- The Type of the return value must be Object.

6.1.7.4 Well-Known Intrinsic Objects

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

Within this specification a reference such as %name% means the intrinsic object, associated with the current realm, corresponding to the name. Determination of the current realm and its intrinsics is described in 8.3. The well-known intrinsics are listed in Table 7.

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Global Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Array%</td>
<td>Array</td>
<td>The Array constructor (22.1.1)</td>
</tr>
<tr>
<td>%ArrayBuffer%</td>
<td>ArrayBuffer</td>
<td>The ArrayBuffer constructor (24.1.2)</td>
</tr>
<tr>
<td>%ArrayBufferPrototype%</td>
<td>ArrayBuffer.prototype</td>
<td>The initial value of the prototype data property of %ArrayBuffer%.</td>
</tr>
<tr>
<td>%ArrayIteratorPrototype%</td>
<td></td>
<td>The prototype of Array iterator objects (22.1.5)</td>
</tr>
<tr>
<td>%ArrayPrototype%</td>
<td>Array.prototype</td>
<td>The initial value of the prototype data property of %Array% (22.1.3)</td>
</tr>
<tr>
<td>%ArrayProto_values%</td>
<td>Array.prototype.values</td>
<td>The initial value of the values data property of %ArrayPrototype% (22.1.3.30)</td>
</tr>
<tr>
<td>%AsyncFunction%</td>
<td></td>
<td>The constructor of async function objects (25.5.1)</td>
</tr>
<tr>
<td>%AsyncFunctionPrototype%</td>
<td></td>
<td>The initial value of the prototype data property of %AsyncFunction%</td>
</tr>
<tr>
<td>%Atomics%</td>
<td>Atomics</td>
<td>The Atomics object (24.4)</td>
</tr>
<tr>
<td>%Boolean%</td>
<td>Boolean</td>
<td>The Boolean constructor (19.3.1)</td>
</tr>
<tr>
<td>%BooleanPrototype%</td>
<td>Boolean.prototype</td>
<td>The initial value of the prototype data property of %Boolean% (19.3.3)</td>
</tr>
<tr>
<td>%DataView%</td>
<td>DataView</td>
<td>The DataView constructor (24.3.2)</td>
</tr>
<tr>
<td>%DataViewPrototype%</td>
<td>DataView.prototype</td>
<td>The initial value of the prototype data property of %DataView%</td>
</tr>
<tr>
<td>%Date%</td>
<td>Date</td>
<td>The Date constructor (20.3.2)</td>
</tr>
<tr>
<td>%DatePrototype%</td>
<td>Date.prototype</td>
<td>The initial value of the prototype data property of %Date%</td>
</tr>
<tr>
<td>%decodeURI%</td>
<td>decodeURI</td>
<td>The decodeURI function (18.2.6.2)</td>
</tr>
<tr>
<td>%decodeURIComponent%</td>
<td>decodeURIComponent</td>
<td>The decodeURIComponent function (18.2.6.3)</td>
</tr>
<tr>
<td>%encodeURI%</td>
<td>encodeURI</td>
<td>The encodeURI function (18.2.6.4)</td>
</tr>
<tr>
<td>%encodeURIComponent%</td>
<td>encodeURIComponent</td>
<td>The encodeURIComponent</td>
</tr>
<tr>
<td>%Error%</td>
<td>Error</td>
<td>The <strong>Error</strong> constructor (19.5.1)</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>%ErrorPrototype%</td>
<td>Error.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Error%</td>
</tr>
<tr>
<td>%eval%</td>
<td>eval</td>
<td>The <strong>eval</strong> function (18.2.1)</td>
</tr>
<tr>
<td>%EvalError%</td>
<td>EvalError</td>
<td>The <strong>EvalError</strong> constructor (19.5.5.1)</td>
</tr>
<tr>
<td>%EvalErrorPrototype%</td>
<td>EvalError.prototype</td>
<td>The initial value of the <strong>prototype</strong> property of %EvalError%</td>
</tr>
<tr>
<td>%Float32Array%</td>
<td>Float32Array</td>
<td>The <strong>Float32Array</strong> constructor (22.2)</td>
</tr>
<tr>
<td>%Float32ArrayPrototype%</td>
<td>Float32Array.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Float32Array%</td>
</tr>
<tr>
<td>%Float64Array%</td>
<td>Float64Array</td>
<td>The <strong>Float64Array</strong> constructor (22.2)</td>
</tr>
<tr>
<td>%Float64ArrayPrototype%</td>
<td>Float64Array.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Float64Array%</td>
</tr>
<tr>
<td>%Function%</td>
<td>Function</td>
<td>The <strong>Function</strong> constructor (19.2.1)</td>
</tr>
<tr>
<td>%FunctionPrototype%</td>
<td>Function.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Function%</td>
</tr>
<tr>
<td>%Generator%</td>
<td></td>
<td>The initial value of the <strong>prototype</strong> property of %GeneratorFunction%</td>
</tr>
<tr>
<td>%GeneratorFunction%</td>
<td></td>
<td>The constructor of generator objects (25.2.1)</td>
</tr>
<tr>
<td>%GeneratorPrototype%</td>
<td></td>
<td>The initial value of the <strong>prototype</strong> property of %GeneratorPrototype%</td>
</tr>
<tr>
<td>%Int8Array%</td>
<td>Int8Array</td>
<td>The <strong>Int8Array</strong> constructor (22.2)</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>%Int8ArrayPrototype%</td>
<td>Int8Array.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Int8Array%</td>
</tr>
<tr>
<td>%Int16Array%</td>
<td>Int16Array</td>
<td>The <strong>Int16Array</strong> constructor (22.2)</td>
</tr>
<tr>
<td>%Int16ArrayPrototype%</td>
<td>Int16Array.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Int16Array%</td>
</tr>
<tr>
<td>%Int32Array%</td>
<td>Int32Array</td>
<td>The <strong>Int32Array</strong> constructor (22.2)</td>
</tr>
<tr>
<td>%Int32ArrayPrototype%</td>
<td>Int32Array.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Int32Array%</td>
</tr>
<tr>
<td>%isFinite%</td>
<td>isFinite</td>
<td>The <strong>isFinite</strong> function (18.2.2)</td>
</tr>
<tr>
<td>%isNaN%</td>
<td>isNaN</td>
<td>The <strong>isNaN</strong> function (18.2.3)</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 <strong>JSON</strong> object (24.5)</td>
</tr>
<tr>
<td>%Map%</td>
<td>Map</td>
<td>The <strong>Map</strong> constructor (23.1.1)</td>
</tr>
<tr>
<td>%MapIteratorPrototype%</td>
<td></td>
<td>The prototype of Map iterator objects (23.1.5)</td>
</tr>
<tr>
<td>%MapPrototype%</td>
<td>Map.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Map%</td>
</tr>
<tr>
<td>%Math%</td>
<td>Math</td>
<td>The <strong>Math</strong> object (20.2)</td>
</tr>
<tr>
<td>%Number%</td>
<td>Number</td>
<td>The <strong>Number</strong> constructor (20.1.1)</td>
</tr>
<tr>
<td>%NumberPrototype%</td>
<td>Number.prototype</td>
<td>The initial value of the</td>
</tr>
<tr>
<td>%NumberPrototype%</td>
<td>Number.prototype</td>
<td>The initial value of the prototype property of %Number%</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>%Object%</td>
<td>Object</td>
<td>The Object constructor (19.1.1)</td>
</tr>
<tr>
<td>%ObjectPrototype%</td>
<td>Object.prototype</td>
<td>The initial value of the prototype data property of %Object%. (19.1.3)</td>
</tr>
<tr>
<td>%ObjProto_toString%</td>
<td>Object.prototype.toString</td>
<td>The initial value of the toString data property of %ObjectPrototype% (19.1.3.6)</td>
</tr>
<tr>
<td>%ObjProto_valueOf%</td>
<td>Object.prototype.valueOf</td>
<td>The initial value of the valueOf data property of %ObjectPrototype% (19.1.3.7)</td>
</tr>
<tr>
<td>%parseFloat%</td>
<td>parseFloat</td>
<td>The parseFloat function (18.2.4)</td>
</tr>
<tr>
<td>%parseInt%</td>
<td>parseInt</td>
<td>The parseInt function (18.2.5)</td>
</tr>
<tr>
<td>%Promise%</td>
<td>Promise</td>
<td>The Promise constructor (25.4.3)</td>
</tr>
<tr>
<td>%PromisePrototype%</td>
<td>Promise.prototype</td>
<td>The initial value of the prototype data property of %Promise%</td>
</tr>
<tr>
<td>%Proxy%</td>
<td>Proxy</td>
<td>The Proxy constructor (26.2.1)</td>
</tr>
<tr>
<td>%RangeError%</td>
<td>RangeError</td>
<td>The RangeError constructor (19.5.5.2)</td>
</tr>
<tr>
<td>%RangeErrorPrototype%</td>
<td>RangeError.prototype</td>
<td>The initial value of the prototype property of %RangeError%</td>
</tr>
<tr>
<td>%ReferenceError%</td>
<td>ReferenceError</td>
<td>The ReferenceError constructor (19.5.5.3)</td>
</tr>
<tr>
<td>%ReferenceErrorPrototype%</td>
<td>ReferenceError.prototype</td>
<td>The initial value of the prototype property of %ReferenceError%</td>
</tr>
<tr>
<td>%Reflect%</td>
<td>Reflect</td>
<td>The <strong>Reflect</strong> object (26.1)</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>%RegExp%</td>
<td>RegExp</td>
<td>The <strong>RegExp</strong> constructor (21.2.3)</td>
</tr>
<tr>
<td>%RegExpPrototype%</td>
<td>RegExp.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %RegExp%</td>
</tr>
<tr>
<td>%Set%</td>
<td>Set</td>
<td>The <strong>Set</strong> constructor (23.2.1)</td>
</tr>
<tr>
<td>%SetIteratorPrototype%</td>
<td></td>
<td>The prototype of Set iterator objects (23.2.5)</td>
</tr>
<tr>
<td>%SetPrototype%</td>
<td>Set.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Set%</td>
</tr>
<tr>
<td>%SharedArrayBuffer%</td>
<td>SharedArrayBuffer</td>
<td>The <strong>SharedArrayBuffer</strong> constructor (24.2.2)</td>
</tr>
<tr>
<td>%SharedArrayBufferPrototype%</td>
<td>SharedArrayBuffer.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %SharedArrayBuffer%</td>
</tr>
<tr>
<td>%String%</td>
<td>String</td>
<td>The <strong>String</strong> constructor (21.1.1)</td>
</tr>
<tr>
<td>%StringIteratorPrototype%</td>
<td></td>
<td>The prototype of String iterator objects (21.1.5)</td>
</tr>
<tr>
<td>%StringPrototype%</td>
<td>String.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %String%</td>
</tr>
<tr>
<td>%Symbol%</td>
<td>Symbol</td>
<td>The <strong>Symbol</strong> constructor (19.4.1)</td>
</tr>
<tr>
<td>%SymbolPrototype%</td>
<td>Symbol.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %Symbol% (19.4.3)</td>
</tr>
<tr>
<td>%SyntaxError%</td>
<td>SyntaxError</td>
<td>The <strong>SyntaxError</strong> constructor (19.5.5.4)</td>
</tr>
<tr>
<td>%SyntaxErrorPrototype%</td>
<td>SyntaxError.prototype</td>
<td>The initial value of the <strong>prototype</strong> data property of %SyntaxError%</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>%ThrowTypeError%</th>
<th>prototype property of %SyntaxError%</th>
<th>A function object that unconditionally throws a new instance of %TypeError%</th>
</tr>
</thead>
<tbody>
<tr>
<td>%TypedArray%</td>
<td></td>
<td>The super class of all typed Array constructors (22.2.1)</td>
</tr>
<tr>
<td>%TypedArrayPrototype%</td>
<td></td>
<td>The initial value of the prototype property of %TypedArray%</td>
</tr>
<tr>
<td>%TypeError%</td>
<td>TypeError</td>
<td>The TypeError constructor (19.5.5.5)</td>
</tr>
<tr>
<td>%TypeErrorPrototype%</td>
<td>TypeError.prototype</td>
<td>The initial value of the prototype property of %TypeError%</td>
</tr>
<tr>
<td>%Uint8Array%</td>
<td>Uint8Array</td>
<td>The Uint8Array constructor (22.2)</td>
</tr>
<tr>
<td>%Uint8ArrayPrototype%</td>
<td>Uint8Array.prototype</td>
<td>The initial value of the prototype data property of %Uint8Array%</td>
</tr>
<tr>
<td>%Uint8ClampedArray%</td>
<td>Uint8ClampedArray</td>
<td>The Uint8ClampedArray constructor (22.2)</td>
</tr>
<tr>
<td>%Uint8ClampedArrayPrototype%</td>
<td>Uint8ClampedArray.prototype</td>
<td>The initial value of the prototype data property of %Uint8ClampedArray%</td>
</tr>
<tr>
<td>%Uint16Array%</td>
<td>Uint16Array</td>
<td>The Uint16Array constructor (22.2)</td>
</tr>
<tr>
<td>%Uint16ArrayPrototype%</td>
<td>Uint16Array.prototype</td>
<td>The initial value of the prototype data property of %Uint16Array%</td>
</tr>
<tr>
<td>%Uint32Array%</td>
<td>Uint32Array</td>
<td>The Uint32Array constructor (22.2)</td>
</tr>
<tr>
<td>%Uint32ArrayPrototype%</td>
<td>Uint32Array.prototype</td>
<td>The initial value of the prototype data property of %Uint32Array%</td>
</tr>
<tr>
<td>%URIError%</td>
<td>URIError</td>
<td>The <strong>URIError</strong> constructor (19.5.5.6)</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>%URIErrorPrototype%</td>
<td>URIError.prototype</td>
<td>The initial value of the <code>prototype</code> property of %URIError%</td>
</tr>
<tr>
<td>%WeakMap%</td>
<td>WeakMap</td>
<td>The <strong>WeakMap</strong> constructor (23.3.1)</td>
</tr>
<tr>
<td>%WeakMapPrototype%</td>
<td>WeakMap.prototype</td>
<td>The initial value of the <code>prototype</code> data property of %WeakMap%</td>
</tr>
<tr>
<td>%WeakSet%</td>
<td>WeakSet</td>
<td>The <strong>WeakSet</strong> constructor (23.4.1)</td>
</tr>
<tr>
<td>%WeakSetPrototype%</td>
<td>WeakSet.prototype</td>
<td>The initial value of the <code>prototype</code> data property of %WeakSet%</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, Lexical Environment, Environment Record, and Data Block. Specification type values are specification artefacts that do not necessarily correspond to any specific entity within an ECMAScript implementation. Specification type values may be used to describe intermediate results of ECMAScript expression evaluation but such values cannot be stored as properties of objects or values of ECMAScript language variables.

#### 6.2.1 The List and Record Specification Types

The List type is used to explain the evaluation of argument lists (see 12.3.6) in `new` expressions, in function calls, and in other algorithms where a simple ordered list of values is needed. Values of the List type are simply ordered sequences of list elements containing the individual values. These sequences may be of any length. The elements of a list may be randomly accessed using 0-origin indices. For notational convenience an array-like syntax can be used to access List elements. For example, `arguments[2]` is shorthand for saying the 3rd element of the List `arguments`. 
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.\([\text{Field2}]\) is shorthand for “the field of R named \([\text{Field2}]\)".

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 conditions.

1. For all \( a, b, \) and \( c \) in \( R \)'s domain:
   a. It is not the case that \( a Ra, \) and
   b. If \( a R b \) and \( b R c \), then \( a Rc \).

   **NOTE 1** The two properties above are called, in order, irreflexivity and transitivity.

A strict total order is a Relation value \( R \) that satisfies the following conditions.
1. For all \( a, b, \) and \( c \) in \( R \)'s domain:
   a. \( a R b \) or \( b R a \), and
   b. It is not the case that \( a R a \), and
   c. If \( a R b \) and \( b R c \), then \( a R c \).

**NOTE 2** The three properties above are called, in order, totality, irreflexivity, and transitivity.

### 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 (\texttt{break}, \texttt{continue}, \texttt{return} and \texttt{throw}) that perform nonlocal transfers of control.

Values of the Completion type are `Record` values whose fields are defined as by Table 8. Such values are referred to as *Completion Records*.

<table>
<thead>
<tr>
<th>Field</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 NormalCompletion

The abstract operation NormalCompletion with a single \textit{argument}, such as:

1. Return \texttt{NormalCompletion(\textit{argument})}.

Is a shorthand that is defined as follows:

1. Return \texttt{Completion{[[Type]]: normal, [[Value]]: \textit{argument}, [[Target]]: empty}}.

#### 6.2.3.2 Implicit Completion Values
The algorithms of this specification often implicitly return Completion Records whose [[Type]] is normal. Unless it is otherwise obvious from the context, an algorithm statement that returns a value that is not a Completion Record, such as:

1. Return "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.

6.2.3.3 Throw an Exception

Algorithms steps that say to throw an exception, such as

1. Throw a TypeError exception.

mean the same things as:

1. Return Completion{[[Type]]: throw, [[Value]]: a newly created TypeError object, [[Target]]: empty}.

6.2.3.4 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, let `argument` be `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, let `hygienicTemp` be `hygienicTemp`.[[Value]].

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

Algorithms steps that say or are otherwise equivalent to:

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

mean the same thing as:

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

### 6.2.3.5 UpdateEmpty (completionRecord, value)

The abstract operation UpdateEmpty with arguments `completionRecord` and `value` performs the following steps:

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

### 6.2.4 The Reference Specification Type

**NOTE** The Reference type is used to explain the behaviour of such operators as `delete`, `typeof`, the assignment operators, the `super` keyword and other language features. For example, the left-hand operand of an assignment is expected to produce a reference.
A Reference is a resolved name or property binding. A Reference consists of three components, the base value component, the referenced name component, and the Boolean-valued strict reference flag. The base value component is either undefined, an Object, a Boolean, a String, a Symbol, a Number, or an Environment Record. A base value component of undefined indicates that the Reference could not be resolved to a binding. The referenced name component is a String or Symbol value.

A Super Reference is a Reference that is used to represents a name binding that was expressed using the super keyword. A Super Reference has an additional thisValue component, and its base value component will never be an Environment Record.

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

- GetBase(V). Returns the base value component of the reference V.
- GetReferencedName(V). Returns the referenced name component of the reference V.
- IsStrictReference(V). Returns the strict reference flag of the reference V.
- HasPrimitiveBase(V). Returns true if Type(V’s base value component) is Boolean, String, Symbol, or Number; otherwise returns false.
- IsPropertyReference(V). Returns true if either the base value component of the reference V is an object or HasPrimitiveBase(V) is true; otherwise returns false.
- IsUnresolvableReference(V). Returns true if the base value component of the reference V is undefined; otherwise returns false.
- IsSuperReference(V). Returns true if the reference V has a thisValue component; otherwise returns false.

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

6.2.4.1 GetValue (V)

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
5. If IsPropertyReference(V) is true, then
   a. If HasPrimitiveBase(V) is true, then
      i. Assert: In this case, base will never be undefined or null.
      ii. Set base to ! ToObject(base).
   b. Return ? base.![Get](GetReferencedName(V), GetThisValue(V)).
5. Else base must be an Environment Record,
   a. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V)) (see 8.1.1).

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

### 6.2.4.2 PutValue (V, W)

1. ReturnIfAbrupt(V).
2. ReturnIfAbrupt(W).
3. If Type(V) is not Reference, throw a ReferenceError exception.
4. Let base be GetBase(V).
5. If IsUnresolvableReference(V) is true, then
   a. If IsStrictReference(V) is true, then
      i. Throw a ReferenceError exception.
   b. Let globalObj be GetGlobalObject().
   c. Return ? Set(globalObj, GetReferencedName(V), W, false).
6. Else if IsPropertyReference(V) is true, then
   a. If HasPrimitiveBase(V) is true, then
      i. Assert: In this case, base will never be undefined or null.
      ii. Set base to ! ToObject(base).
   b. Let succeeded be ? base.[[Set]](GetReferencedName(V), W, GetThisValue(V)).
   c. If succeeded is false and IsStrictReference(V) is true, throw a TypeError exception.
   d. Return.
7. Else base must be an Environment Record,
   a. Return ? base.SetMutableBinding(GetReferencedName(V), W, IsStrictReference(V)) (see 8.1.1).

**NOTE** The object that may be created in step 6.a.ii is not accessible outside of the above algorithm and the ordinary object [[Set]] internal method. An implementation might choose to avoid the actual creation of that object.

### 6.2.4.3 GetThisValue (V)

1. Assert: IsPropertyReference(V) is true.
2. If IsSuperReference(V) is true, then
   a. Return the value of the thisValue component of the reference V.

### 6.2.4.4 InitializeReferencedBinding (V, W)

1. ReturnIfAbrupt(V).
2. ReturnIfAbrupt(W).
3. Assert: Type(V) is Reference.
4. Assert: IsUnresolvableReference(V) is false.
5. Let base be GetBase(V).
6. Assert: \textit{base} is an \textit{Environment Record}.

7. Return \textit{base}.InitializeBinding(\textit{GetReferencedName}(V), \textit{W}).

### 6.2.5 The Property Descriptor Specification Type

The \textit{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 \texttt{[[Value]]} or \texttt{[[Writable]]}. An accessor Property Descriptor is one that includes any fields named either \texttt{[[Get]]} or \texttt{[[Set]]}. Any Property Descriptor may have fields named \texttt{[[Enumerable]]} and \texttt{[[Configurable]]}. A Property Descriptor value may not be both a data Property Descriptor and an accessor Property Descriptor; however, it may be neither. A generic Property Descriptor is a Property Descriptor value that is neither a data Property Descriptor nor an accessor Property Descriptor. A fully populated Property Descriptor is one that is either an accessor Property Descriptor or a data Property Descriptor and that has all of the fields that correspond to the property attributes defined in either Table 2 or Table 3.

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

#### 6.2.5.1 IsAccessorDescriptor (\textit{Desc})

When the abstract operation IsAccessorDescriptor is called with Property Descriptor \textit{Desc}, the following steps are taken:

1. If \textit{Desc} is \texttt{undefined}, return \texttt{false}.
2. If both \textit{Desc}.[[Get]] and \textit{Desc}.[[Set]] are absent, return \texttt{false}.
3. Return \texttt{true}.

#### 6.2.5.2 IsDataDescriptor (\textit{Desc})

When the abstract operation IsDataDescriptor is called with Property Descriptor \textit{Desc}, the following steps are taken:

1. If \textit{Desc} is \texttt{undefined}, return \texttt{false}.
2. If both \textit{Desc}.[[Value]] and \textit{Desc}.[[Writable]] are absent, return \texttt{false}.
3. Return \texttt{true}. 
6.2.5.3 IsGenericDescriptor (Desc)

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

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

6.2.5.4 FromPropertyDescriptor (Desc)

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

1. If Desc is undefined, return undefined.
2. Let obj be ObjectCreate(%ObjectPrototype%).
3. Assert: obj is an extensible ordinary object with no own properties.
4. If Desc has a [[Value]] field, then
   a. Perform CreateDataProperty(obj, "value", Desc.[[Value]]).
5. If Desc has a [[Writable]] field, then
   a. Perform CreateDataProperty(obj, "writable", Desc.[[Writable]]).
6. If Desc has a [[Get]] field, then
   a. Perform CreateDataProperty(obj, "get", Desc.[[Get]]).
7. If Desc has a [[Set]] field, then
   a. Perform CreateDataProperty(obj, "set", Desc.[[Set]]).
8. If Desc has an [[Enumerable]] field, then
   a. Perform CreateDataProperty(obj, "enumerable", Desc.[[Enumerable]]).
9. If Desc has a [[Configurable]] field, then
   a. Perform CreateDataProperty(obj, "configurable", Desc.[[Configurable]]).
10. Assert: All of the above CreateDataProperty operations return true.
11. Return obj.

6.2.5.5 ToPropertyDescriptor (Obj)

When the abstract operation ToPropertyDescriptor is called with object Obj, the following steps are taken:

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 enum be ToBoolean(? Get(Obj, "enumerable")).
   b. Set desc.[[Enumerable]] to enum.
5. Let $hasConfigurable$ be $\text{HasProperty}(\text{Obj}, \text{"configurable"})$.

6. If $hasConfigurable$ is true, then
   a. Let $conf$ be $\text{ToBoolean}?(\text{Get}(\text{Obj}, \text{"configurable"}))$.
   b. Set $\text{desc}[[\text{Configurable}]]$ to $conf$.

7. Let $hasValue$ be $\text{HasProperty}(\text{Obj}, \text{"value"})$.

8. If $hasValue$ is true, then
   a. Let $value$ be $\text{Get}(\text{Obj}, \text{"value"})$.
   b. Set $\text{desc}[[\text{Value}]]$ to $value$.

9. Let $hasWritable$ be $\text{HasProperty}(\text{Obj}, \text{"writable"})$.

10. If $hasWritable$ is true, then
    a. Let $writable$ be $\text{ToBoolean}?(\text{Get}(\text{Obj}, \text{"writable"}))$.
    b. Set $\text{desc}[[\text{Writable}]]$ to $writable$.

11. Let $hasGet$ be $\text{HasProperty}(\text{Obj}, \text{"get"})$.

12. If $hasGet$ is true, then
    a. Let $getter$ be $\text{Get}(\text{Obj}, \text{"get"})$.
    b. If $\text{IsCallable}(getter)$ is false and $getter$ is not undefined, throw a $\text{TypeError}$ exception.
    c. Set $\text{desc}[[\text{Get}]]$ to $getter$.

13. Let $hasSet$ be $\text{HasProperty}(\text{Obj}, \text{"set"})$.

14. If $hasSet$ is true, then
    a. Let $setter$ be $\text{Get}(\text{Obj}, \text{"set"})$.
    b. If $\text{IsCallable}(setter)$ is false and $setter$ is not undefined, throw a $\text{TypeError}$ exception.
    c. Set $\text{desc}[[\text{Set}]]$ to $setter$.

15. If $\text{desc}[[\text{Get}]]$ is present or $\text{desc}[[\text{Set}]]$ is present, then
    a. If $\text{desc}[[\text{Value}]]$ is present or $\text{desc}[[\text{Writable}]]$ is present, throw a $\text{TypeError}$ exception.

16. Return $\text{desc}$.

### 6.2.5.6 CompletePropertyDescriptor ($Desc$)

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

1. Assert: $Desc$ is a Property Descriptor.

2. Let $like$ be $\text{Record}([[\text{Value}]]: \text{undefined}, [[\text{Writable}]]: \text{false}, [[\text{Get}]]: \text{undefined}, [[\text{Set}]]: \text{undefined}, [[\text{Enumerable}]]: \text{false}, [[\text{Configurable}]]: \text{false})$.

3. If $\text{IsGenericDescriptor}(Desc)$ is true or $\text{IsDataDescriptor}(Desc)$ 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 Lexical Environment and Environment Record Specification Types

The Lexical Environment and Environment Record types are used to explain the behaviour of name resolution in nested functions and blocks. These types and the operations upon them are defined in 8.1.

6.2.7 Data Blocks

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

For notational convenience within this specification, an array-like syntax can be used to access the individual bytes of a Data Block value. This notation presents a Data Block value as a 0-originated 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:

6.2.7.1 CreateByteDataBlock (size)

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

1. Assert: size≥0.
2. Let \( db \) be a new Data Block value consisting of \( size \) bytes. If it is impossible to create such a Data Block, throw a RangeError exception.

3. Set all of the bytes of \( db \) to 0.

4. Return \( db \).

### 6.2.7.2 CreateSharedByteDataBlock(\( size \))

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

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

### 6.2.7.3 CopyDataBlockBytes(\( toBlock, toIndex, fromBlock, fromIndex, count \))

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

1. Assert: \( fromBlock \) and \( toBlock \) are distinct Data Block or Shared Data Block values.
2. Assert: \( fromIndex, toIndex, \) and \( count \) are integer values \( \geq 0 \).
3. Let \( fromSize \) be the number of bytes in \( fromBlock \).
4. Assert: \( fromIndex + count \leq fromSize \).
5. Let \( toSize \) be the number of bytes in \( toBlock \).
6. Assert: \( toIndex + count \leq toSize \).
7. Repeat, while \( count > 0 \)
   a. 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 \( execution \).[[EventLists]] whose [[AgentSignifier]] is AgentSignifier().
      iii. Let \( bytes \) be a List of length 1 that contains 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 WriteSharedMemory{ [[Order]]: "Unordered", [[NoTear]]: true, [[Block]]: toBlock, [[ByteIndex]]: toIndex, [[ElementSize]]: 1, [[Payload]]: bytes } to eventList.

viii. Append Chosen Value Record { [[Event]]: readEvent, [[ChosenValue]]: bytes } to execution. [[ChosenValues]].

b. Otherwise, set toBlock[toIndex] to fromBlock[fromIndex].

c. Increment toIndex and fromIndex each by 1.

d. Decrement count by 1.

8. Return NormalCompletion( empty).

7 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification.

7.1 Type Conversion

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.

7.1.1 ToPrimitive ( input [, PreferredType ] )

The abstract operation ToPrimitive takes an input argument and an optional argument PreferredType. The abstract operation ToPrimitive converts its input argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint PreferredType to favour that type. Conversion occurs according to the following algorithm:

1. Assert: input is an ECMAScript language value.
2. If Type(input) is Object, then
   a. If PreferredType was not passed, let hint be "default".
   b. Else if PreferredType is hint String, let hint be "string".
   c. Else PreferredType is hint Number, let hint be "number".

d. Let $exoticToPrim$ be $\textbf{false}$ $\text{GetMethod}(input, @@toPrimitive)$.

e. If $exoticToPrim$ is not $\textbf{undefined}$, then
   i. Let $result$ be $\text{Call}(exoticToPrim, input, « hint »)$.
   ii. If $\text{Type}(result)$ is not Object, return $result$.
   iii. Throw a $\textbf{TypeError}$ exception.
   f. If $hint$ is "default", set $hint$ to "number".

g. Return $\textbf{false}$ $\text{OrdinaryToPrimitive}(input, hint)$.

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 20.3.4.45) and
Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects
treat no hint as if the hint were String.

### 7.1.1.1 OrdinaryToPrimitive $(O, hint)$

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

1. Assert: $\text{Type}(O)$ is Object.
2. Assert: $\text{Type}(hint)$ is String and its value 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 $name$ in $methodNames$ in List order, do
   a. Let $method$ be ? $\text{Get}(O, name)$.
   b. If $\text{IsCallable}(method)$ is true, then
      i. Let $result$ be ? $\text{Call}(method, O)$.
      ii. If $\text{Type}(result)$ is not Object, return $result$.
6. Throw a $\textbf{TypeError}$ exception.

### 7.1.2 ToBoolean $(argument)$

The abstract operation ToBoolean converts $argument$ to a value of type Boolean according to Table 9:

**Table 9: ToBoolean Conversions**

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Return $\textbf{false}$</td>
</tr>
</tbody>
</table>
null

RETURN false.

BOOLEAN

RETURN argument.

NUMBER

If argument is +0, -0, or NaN, return false; otherwise return true.

STRING

If argument is the empty String (its length is zero), return false; otherwise return true.

SYMBOL

RETURN true.

OBJECT

RETURN true.

The abstract operation ToNumber converts argument to a value of type Number according to Table 10:

### Table 10: 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 argument is true, return 1. If argument is false, return +0.</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>STRING</td>
<td>See grammar and conversion algorithm below.</td>
</tr>
<tr>
<td>SYMBOL</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>OBJECT</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1. Let primValue be ? ToPrimitive(argument, hint Number).</td>
</tr>
<tr>
<td></td>
<td>2. Return ? ToNumber(primValue).</td>
</tr>
</tbody>
</table>

#### 7.1.3.1 ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points (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 Unicode BMP code points so the result will be NaN if the string contains the UTF-16 encoding of any supplementary code points or any unpaired surrogate code points.

Syntax

```
StringNumericLiteral  ::=  
  StrWhiteSpace_opt  
  StrWhiteSpace_opt  StrNumericLiteral  StrWhiteSpace_opt

StrWhiteSpace  ::=  
  StrWhiteSpaceChar  StrWhiteSpace_opt

StrWhiteSpaceChar  ::=  
  WhiteSpace  
  Line Terminator

StrNumericLiteral  ::=  
  StrDecimalLiteral  
  BinaryIntegerLiteral  
  OctalIntegerLiteral  
  HexIntegerLiteral

StrDecimalLiteral  ::=  
  StrUnsignedDecimalLiteral  
  +  StrUnsignedDecimalLiteral  
  -  StrUnsignedDecimalLiteral

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

All grammar symbols not explicitly defined above have the definitions used in the Lexical Grammar for numeric literals (11.8.3)

NOTE 2 Some differences should be noted between the syntax of a StringNumericLiteral and a NumericLiteral:

- A StringNumericLiteral may include leading and/or trailing white space and/or line terminators.
- A StringNumericLiteral that is decimal may have any number of leading 0 digits.
• A \texttt{StringNumericLiteral} that is decimal may include a + or - to indicate its sign.
• A \texttt{StringNumericLiteral} that is empty or contains only white space is converted to \texttt{+0}.
• \texttt{Infinity} and \texttt{-Infinity} are recognized as a \texttt{StringNumericLiteral} but not as a \texttt{NumericLiteral}.

7.1.3.1.1 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 11.8.3), but some of the details are different, so the process for converting a String numeric literal to a value of Number type is given here. 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 11.8.3.1.

- The MV of \texttt{StringNumericLiteral} :: [empty] is 0.
- The MV of \texttt{StringNumericLiteral} :: \texttt{ StrWhiteSpace} is 0.
- The MV of \texttt{StringNumericLiteral} :: \texttt{ StrWhiteSpace StringLiteral StrWhiteSpace} is the MV of \texttt{StringLiteral}, no matter whether white space is present or not.
- The MV of \texttt{StringLiteral} :: \texttt{StrDecimalLiteral} is the MV of \texttt{StrDecimalLiteral}.
- The MV of \texttt{StringLiteral} :: \texttt{BinaryIntegerLiteral} is the MV of \texttt{BinaryIntegerLiteral}.
- The MV of \texttt{StringLiteral} :: \texttt{OctalIntegerLiteral} is the MV of \texttt{OctalIntegerLiteral}.
- The MV of \texttt{StringLiteral} :: \texttt{HexIntegerLiteral} is the MV of \texttt{HexIntegerLiteral}.
- The MV of \texttt{StrDecimalLiteral} :: \texttt{unsignedDecimalLiteral} is the MV of \texttt{unsignedDecimalLiteral}.
- The MV of \texttt{StrDecimalLiteral} :: + \texttt{unsignedDecimalLiteral} is the MV of \texttt{unsignedDecimalLiteral}.
- The MV of \texttt{StrDecimalLiteral} :: - \texttt{unsignedDecimalLiteral} is the negative of the MV of \texttt{unsignedDecimalLiteral}. (Note that if the MV of \texttt{unsignedDecimalLiteral} 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 \texttt{+0} or \texttt{-0} as appropriate.)
- The MV of \texttt{unsignedDecimalLiteral} :: \texttt{Infinity} is \(10^{10000}\) (a value so large that it will round to \(+\infty\)).
- The MV of \texttt{unsignedDecimalLiteral} :: \texttt{DecimalDigits} is the MV of \texttt{DecimalDigits}.
- The MV of \texttt{unsignedDecimalLiteral} :: \texttt{DecimalDigits} \texttt{DecimalDigits} is the MV of the first \texttt{DecimalDigits} plus (the MV of the second \texttt{DecimalDigits} times \(10^{-n}\)), where \(n\) is the number of code points in the second \texttt{DecimalDigits}.
- The MV of \texttt{unsignedDecimalLiteral} :: \texttt{DecimalDigits} \texttt{ExponentPart} is the MV of \texttt{DecimalDigits} times \(10^{e}\), where \(e\) is the MV of \texttt{ExponentPart}.
- The MV of \texttt{unsignedDecimalLiteral} :: \texttt{DecimalDigits} \texttt{ExponentPart} is (the MV of the first \texttt{DecimalDigits} plus (the MV of the second \texttt{DecimalDigits} times \(10^{-n}\))) times \(10^{e}\), where \(n\) is the number of code points in the second \texttt{DecimalDigits} and \(e\) is the MV of \texttt{ExponentPart}. 
The MV of \( \text{StrUnsignedDecimalLiteral} \) :: \( \text{DecimalDigits} \) is the MV of \( \text{DecimalDigits} \) times \( 10^{-n} \), where \( n \) is the number of code points in \( \text{DecimalDigits} \).

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

The MV of \( \text{StrUnsignedDecimalLiteral} \) :: \( \text{DecimalDigits} \ \text{ExponentPart} \) is the MV of \( \text{DecimalDigits} \) times \( 10^{e} \), where \( e \) 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), 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 \( \text{ExponentPart} \) and

- it is not \( 0 \); or
- there is a nonzero digit to its left and there is a nonzero digit, not in the \( \text{ExponentPart} \), to its right.

### 7.1.4 ToInteger ( \text{argument} )

The abstract operation ToInteger converts \( \text{argument} \) to an integral numeric value. This abstract operation functions as follows:

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

### 7.1.5 ToInt32 ( \text{argument} )

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

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

**NOTE**

Given the above definition of ToInt32:

- The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- \( \text{ToInt32}(\text{ToUint32}(x)) \) is equal to \( \text{ToInt32}(x) \) 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.

### 7.1.6 ToUint32 (\textit{argument})

The abstract operation ToUint32 converts \textit{argument} to one of \( 2^{32} \) integer values in the range 0 through \( 2^{32}-1 \), inclusive. This abstract operation functions as follows:

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

**NOTE**

Given the above definition of ToUint32:

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

### 7.1.7 ToInt16 (\textit{argument})

The abstract operation ToInt16 converts \textit{argument} to one of \( 2^{16} \) integer values in the range -32768 through 32767, inclusive. This abstract operation functions as follows:

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

### 7.1.8 ToUint16 ( \textit{argument} )

The abstract operation ToUint16 converts argument to one of 2^{16} integer values in the range 0 through 2^{16}-1, inclusive. This abstract operation functions as follows:

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

**Note**

Given the above definition of ToUint16:

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

### 7.1.9 ToInt8 ( \textit{argument} )

The abstract operation ToInt8 converts argument to one of 2^{8} integer values in the range -128 through 127, inclusive. This abstract operation functions as follows:

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

### 7.1.10 ToUint8 ( \textit{argument} )

The abstract operation ToUint8 converts argument to one of 2^{8} integer values in the range 0 through 255, inclusive. This abstract operation functions as follows:
1. Let `number` be `ToNumber(\text{argument})`.
2. If `number` is `\text{NaN}`, `-0`, `-\infty`, or `-\infty`, return `+0`.
3. Let `int` be the mathematical value that is the same sign as `number` and whose magnitude is `\text{floor(abs(number))}`.
4. Let `int8bit` be `int` modulo $2^8$.
5. Return `int8bit`.

### 7.1.11 ToUint8Clamp (argument)

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

1. Let `number` be `ToNumber(\text{argument})`.
2. If `number` is `\text{NaN}`, return `+0`.
3. If `number \leq 0`, return `+0`.
4. If `number \geq 255`, return 255.
5. Let `f` be `\text{floor(number)}`.
6. If `f + 0.5 < number`, return `f + 1`.
7. If `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-integer values and does not convert $+\infty$ to 0. ToUint8Clamp does “round half to even” tie-breaking. This differs from `\text{Math.round}` which does “round half up” tie-breaking.

### 7.1.12 ToString (argument)

The abstract operation ToString converts `\text{argument}` to a value of type String according to \textbf{Table 11}:

<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 <code>\text{argument}</code> is \textbf{true}, return &quot;true&quot;.</td>
</tr>
<tr>
<td></td>
<td>If <code>\text{argument}</code> is \textbf{false}, return &quot;false&quot;.</td>
</tr>
</tbody>
</table>
### 7.1.12.1 ToString Applied to the Number Type

The abstract operation `ToString` converts a Number \( m \) to String format as follows:

1. If \( m \) is `NaN`, return the String "NaN".
2. If \( m \) is `+0` or `-0`, return the String "0".
3. If \( m \) is less than zero, return the String concatenation of the String "-" and !`ToString(-m)`.
4. If \( m \) is `+\infty`, return the String "Infinity".
5. Otherwise, let \( n, k, \) and \( s \) be integers such that \( k \geq 1, 10^{k-1} \leq s < 10^k \), the Number value for \( s \times 10^{n-k} \) is \( m \), and \( k \) is as small as possible. Note that \( k \) is the number of digits in the decimal representation of \( s \), that \( s \) is not divisible by 10, and that the least significant digit of \( s \) is not necessarily uniquely determined by these criteria.
6. If \( k \leq n \leq 21 \), return the String consisting of the code units of the \( k \) digits of the decimal representation of \( s \) (in order, with no leading zeroes), followed by \( n-k \) occurrences of the code unit `0x0030` (DIGIT ZERO).
7. If \( 0 < n \leq 21 \), return the String consisting of the code units of the most significant \( n \) digits of the decimal representation of \( s \), followed by the code unit `0x002E` (FULL STOP), followed by the code units of the remaining \( k-n \) digits of the decimal representation of \( s \).
8. If \( -6 < n \leq 0 \), return the String consisting of the code unit `0x0030` (DIGIT ZERO), followed by the code unit `0x002E` (FULL STOP), followed by \( -n \) occurrences of the code unit `0x0030` (DIGIT ZERO), followed by the code units of the \( k \) digits of the decimal representation of \( s \).
9. Otherwise, if \( k = 1 \), return the String consisting of the code unit of the single digit of \( s \), followed by code unit `0x0065` (LATIN SMALL LETTER E), followed by the code unit `0x002B` (PLUS SIGN) or the code unit `0x002D` (HYPHEN-MINUS) according to whether \( n-1 \) is positive or negative, followed by the code units of the decimal representation of the integer `abs(n-1)` (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of \( s \), followed by code unit `0x002E` (FULL STOP), followed by the code units of the remaining \( k-1 \) digits of the decimal representation of \( s \), followed by code unit `0x0065` (LATIN SMALL
LETTER E, followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether \( n-1 \) is positive or negative, followed by 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, then \( \text{ToNumber}(\text{ToString}(x)) \) is exactly the same Number value as \( x \).
- The least significant digit of \( s \) is not always uniquely determined by the requirements listed in step 5.

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

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

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


### 7.1.13 ToObject (argument)

The abstract operation ToObject converts argument to a value of type Object according to Table 12:

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception</td>
</tr>
</tbody>
</table>

**Table 12: ToObject Conversions**
### 7.1.14 ToPropertyKey (argument)

The abstract operation ToPropertyKey converts `argument` to a value that can be used as a property key by performing the following steps:

1. Let `key` be `? ToPrimitive(argument, hint String)`.
2. If `Type(key)` is Symbol, then
   a. Return `key`.
3. Return `! ToString(key)`.

### 7.1.15 ToLength (argument)

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

1. Let `len` be `? ToInteger(argument)`.
2. If `len ≤ +0`, return `+0`.
3. Return `min(len, 2^{53} - 1)`.

### 7.1.16 CanonicalNumericIndexString (argument)

The abstract operation CanonicalNumericIndexString returns `argument` converted to a numeric value if it is a String representation of a Number that would be produced by `ToString`, or the string "-0". Otherwise, it returns `undefined`. This abstract operation functions as follows:
1. Assert: `Type(argument)` is String.
2. If `argument` is "-0", return -0.
3. Let \( n \) be `ToNumber(argument)`.
4. If `SameValue(! ToString(n), argument)` is false, return undefined.
5. Return \( n \).

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

### 7.1.17 ToIndex ( value )

The abstract operation ToIndex returns `value` argument converted to a numeric value if it is a valid integer index value. This abstract operation functions as follows:

1. If `value` is undefined, then
   a. Let \( index \) be 0.
2. Else,
   a. Let \( integerIndex \) be ? ToInteger(`value`).
   b. If `integerIndex` < 0, throw a RangeError exception.
   c. Let \( index \) be `! ToLength(integerIndex)`.
   d. If `SameValueZero(integerIndex, index)` is false, throw a RangeError exception.
3. Return \( index \).

### 7.2 Testing and Comparison Operations

#### 7.2.1 RequireObjectCoercible ( argument )

The abstract operation RequireObjectCoercible throws an error if `argument` is a value that cannot be converted to an Object using ToObject. It is defined by Table 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 <code>argument</code>.</td>
</tr>
<tr>
<td>Number</td>
<td>Return <code>argument</code>.</td>
</tr>
<tr>
<td>String</td>
<td>Return <code>argument</code>.</td>
</tr>
</tbody>
</table>
### 7.2.2 IsArray (argument)

The abstract operation IsArray takes one argument `argument`, and performs the following steps:

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

### 7.2.3 IsCallable (argument)

The abstract operation IsCallable determines if `argument`, which must be an ECMAScript language value, is a callable function with a `[[Call]]` internal method.

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 determines if `argument`, which must be an ECMAScript language value, is a function object with a `[[Construct]]` internal method.

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

1. Assert: `Type(O)` is Object.
2. Return \( O.([[\text{IsExtensible}]]()) \).

### 7.2.6 IsInteger (\textit{argument} )

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

1. If \( \text{Type}(\text{argument}) \) is not Number, return \textit{false}.
2. If \( \text{argument} \) is NaN, \(+\infty\), or \(-\infty\), return \textit{false}.
3. If \( \text{floor}(\text{abs}(\text{argument})) \neq \text{abs}(\text{argument}) \), return \textit{false}.
4. Return \textit{true}.

### 7.2.7 IsPropertyKey (\textit{argument} )

The abstract operation IsPropertyKey determines if \( \text{argument} \), which must be an ECMAScript language value, is a value that may be used as a property key.

1. If \( \text{Type}(\text{argument}) \) is String, return \textit{true}.
2. If \( \text{Type}(\text{argument}) \) is Symbol, return \textit{true}.
3. Return \textit{false}.

### 7.2.8 IsRegExp (\textit{argument} )

The abstract operation IsRegExp with argument \( \text{argument} \) performs the following steps:

1. If \( \text{Type}(\text{argument}) \) is not Object, return \textit{false}.
2. Let \( \text{isRegExp} \) be \( ? \text{Get}(\text{argument}, @@\text{match}) \).
3. If \( \text{isRegExp} \) is not \textit{undefined}, return \textit{ToBoolean}(\text{isRegExp})
4. If \( \text{argument} \) has a \([[[\text{RegExpMatcher}]]] \) internal slot, return \textit{true}.
5. Return \textit{false}.

### 7.2.9 SameValue (\( x, y \))

The internal comparison abstract operation SameValue(\( x, y \)), where \( x \) and \( y \) are ECMAScript language values, produces \textit{true} or \textit{false}. Such a comparison is performed as follows:

1. If \( \text{Type}(x) \) is different from \( \text{Type}(y) \), return \textit{false}.
2. If \( \text{Type}(x) \) is Number, then
   a. If \( x \) is NaN and \( y \) is NaN, return \textit{true}.
   b. If \( x \) is \(+0\) and \( y \) is \(-0\), return \textit{false}.
   c. If \( x \) is \(-0\) and \( y \) is \(+0\), return \textit{false}.
   d. If \( x \) is the same Number value as \( y \), return \textit{true}. 

3. If \( x \) is not Number and \( y \) is Number, then
   a. If \( x \) is NaN, return \textit{false}.
   b. If \( y \) is NaN, return \textit{false}.
   c. If \( x \) is \(+\infty\) and \( y \) is \(-\infty\), return \textit{false}.
   d. If \( x \) is \(+\infty\) and \( y \) is \(+\infty\), return \textit{true}.
   e. If \( x \) is \(-\infty\) and \( y \) is \(+\infty\), return \textit{false}.
   f. If \( x \) is \(-\infty\) and \( y \) is \(-\infty\), return \textit{true}.
4. return true.
e. Return false.

3. Return SameValueNonNumber(x, y).

NOTE This algorithm differs from the Strict Equality Comparison Algorithm in its treatment of signed zeroes and NaNs.

7.2.10 SameValueZero (x, y)

The internal comparison abstract operation SameValueZero(x, y), where x and y are ECMAScript language values, produces true or false. Such a comparison is performed as follows:

1. If Type(x) is different from Type(y), return false.
2. If Type(x) is Number, then
   a. If x is NaN and y is NaN, return true.
   b. If x is +0 and y is -0, return true.
   c. If x is -0 and y is +0, return true.
   d. If x is the same Number value as y, return true.
   e. Return false.
3. Return SameValueNonNumber(x, y).

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

7.2.11 SameValueNonNumber (x, y)

The internal comparison abstract operation SameValueNonNumber(x, y), where neither x nor y are Number values, produces true or false. Such a comparison is performed as follows:

1. Assert: Type(x) is not Number.
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.

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

1. If the \textit{LeftFirst} flag is \texttt{true}, then
   a. Let \( px \) be ? \texttt{ToPrimitive}(\( x \), hint Number).
   b. Let \( py \) be ? \texttt{ToPrimitive}(\( y \), hint Number).
2. Else the order of evaluation needs to be reversed to preserve left to right evaluation,
   a. Let \( py \) be ? \texttt{ToPrimitive}(\( y \), hint Number).
   b. Let \( px \) be ? \texttt{ToPrimitive}(\( x \), hint Number).
3. If both \( px \) and \( py \) are Strings, then
   a. If \( py \) is a prefix of \( px \), return \texttt{false}. (A String value \( p \) is a prefix of String value \( q \) if \( q \) can be the result of concatenating \( p \) and some other String \( r \). Note that any String is a prefix of itself, because \( r \) may be the empty String.)
   b. If \( px \) is a prefix of \( py \), return \texttt{true}.
   c. Let \( k \) be the smallest nonnegative integer such that the 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 code unit value at index \( k \) within \( px \).
   e. Let \( n \) be the integer that is the code unit value at index \( k \) within \( py \).
   f. If \( m < n \), return \texttt{true}. Otherwise, return \texttt{false}.
4. Else,
   a. Let \( nx \) be ? \texttt{ToNumber}(\( px \)). Because \( px \) and \( py \) are primitive values evaluation order is not important.
   b. Let \( ny \) be ? \texttt{ToNumber}(\( py \)).
   c. If \( nx \) is \texttt{NaN}, return \texttt{undefined}.
   d. If \( ny \) is \texttt{NaN}, return \texttt{undefined}.
   e. If \( nx \) and \( ny \) are the same Number value, return \texttt{false}.
   f. If \( nx \) is \texttt{+0} and \( ny \) is \texttt{-0}, return \texttt{false}.
   g. If \( nx \) is \texttt{-0} and \( ny \) is \texttt{+0}, return \texttt{false}.
   h. If \( nx \) is \texttt{+\infty}, return \texttt{false}.
   i. If \( ny \) is \texttt{+\infty}, return \texttt{true}.
   j. If \( ny \) is \texttt{-\infty}, return \texttt{false}.
   k. If \( nx \) is \texttt{-\infty}, return \texttt{true}.
   l. If the mathematical value of \( nx \) is less than the mathematical value of \( ny \) —note that these mathematical values are both finite and not both zero—return \texttt{true}. Otherwise, return \texttt{false}.
Step 3 differs from step 7 in the algorithm for the addition operator \(+\) (12.8.3) in using “and” instead of “or”.

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.

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

### 7.2.14 Strict Equality Comparison

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

1. If \(\text{Type}(x)\) is different from \(\text{Type}(y)\), return **false**.
2. If \(\text{Type}(x)\) is Number, then
   a. If \(x\) is **NaN**, return **false**.
   b. If \(y\) is **NaN**, return **false**.
   c. If \(x\) is the same Number value as \(y\), return **true**.
d. If \( x \) is +0 and \( y \) is -0, return true.
e. If \( x \) is -0 and \( y \) is +0, return true.
f. Return false.

3. Return SameValueNonNumber\((x, y)\).

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

7.3 Operations on Objects

7.3.1 Get \((O, P)\)

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

1. Assert: Type\((O)\) is Object.
2. Assert: IsPropertyKey\((P)\) is true.
3. Return ? \( O.[[\text{Get}]](P, O) \).

7.3.2 GetV \((V, P)\)

The abstract operation GetV 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. The operation is called with arguments \( V \) and \( P \) where \( V \) is the value and \( P \) is the property key. This abstract operation performs the following steps:

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

7.3.3 Set \((O, P, V, \text{Throw})\)

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

1. Assert: Type\((O)\) is Object.
2. Assert: IsPropertyKey\((P)\) is true.
3. Assert: Type\((Throw)\) is Boolean.
Let `success` be \( O[[\text{Set}]](P, V, O) \).

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

Return `success`.

### 7.3.4 CreateDataProperty \((O, P, V)\)

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

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

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

### 7.3.5 CreateMethodProperty \((O, P, V)\)

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

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

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

### 7.3.6 CreateDataPropertyOrThrow \((O, P, V)\)


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

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

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

### 7.3.7 DefinePropertyOrThrow ( $O$, $P$, $desc$ )

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

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

### 7.3.8 DeletePropertyOrThrow ( $O$, $P$ )

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

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

### 7.3.9 GetMethod ( V, P )

The abstract operation GetMethod 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. The operation is called with arguments *V* and *P* where *V* is the *ECMAScript language value*, *P* is the property key. This abstract operation performs the following steps:

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`.

### 7.3.10 HasProperty ( O, P )

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

1. Assert: `Type(O)` is Object.
2. Assert: `IsPropertyKey(P)` is **true**.

### 7.3.11 HasOwnProperty ( O, P )

The abstract operation HasOwnProperty is used to determine whether an object has an own property with the specified property key. A Boolean value is returned. The operation is called with arguments *O* and *P* where *O* is the object and *P* is the property key. This abstract operation performs the following steps:

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

### 7.3.12 Call ( F, V [ , argumentsList ] )
The abstract operation Call is used to call the [[Call]] internal method of a function object. The operation is called with arguments \( F, V \), and optionally \( argumentsList \) where \( F \) is the function object, \( V \) is an ECMAScript language value that is the \( this \) value of the [[Call]], and \( argumentsList \) is the value passed to the corresponding argument of the internal method. If \( argumentsList \) is not present, a new empty List is used as its value. This abstract operation performs the following steps:

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

### 7.3.13 Construct ( \( F \), \( argumentsList \) [, \( newTarget \) ] )

The abstract operation Construct is used to call the [[Construct]] internal method of a function object. The operation is called with arguments \( F \), and optionally \( argumentsList \), and \( newTarget \) where \( F \) is the function object. \( argumentsList \) and \( newTarget \) are the values to be passed as the corresponding arguments of the internal method. If \( argumentsList \) is not present, a new empty List is used as its value. If \( newTarget \) is not present, \( F \) is used as its value. This abstract operation performs the following steps:

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

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

### 7.3.14 SetIntegrityLevel ( \( O, level \) )

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

1. Assert: \( Type(O) \) is Object.
2. Assert: \( level \) is either "sealed" or "frozen".
3. Let \( status \) 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 \( level \) is "frozen",
   a. For each element \( k \) of \( keys \), do
Let currentDesc be \( O.[[\text{GetOwnProperty}]](k) \).

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 \( \text{DefinePropertyOrThrow}(O, k, \text{desc}) \).

8. Return true.

### 7.3.15 TestIntegrityLevel (\( O, \text{level} \) )

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

1. Assert: Type(\( O \)) is Object.
2. Assert: level is either "sealed" or "frozen".
3. Let status be ? IsExtensible(\( O \)).
4. If status is true, return false.
5. NOTE: If the object is extensible, none of its properties are examined.
6. Let keys be \( O.[[\text{OwnPropertyKeys}]]() \).
7. For each element \( k \) of keys, do
   a. Let currentDesc be \( O.[[\text{GetOwnProperty}]](k) \).
   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.

### 7.3.16 CreateArrayFromList (\( \text{elements} \) )

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

1. Assert: elements is a List whose elements are all ECMAScript language values.
2. Let array be ! ArrayCreate(0).
3. Let \( n \) be 0.
4. For each element \( e \) of elements, do
   a. Let status be CreateDataProperty(array, ! ToString(\( n \)), e).
   b. Assert: status is true.
   c. Increment \( n \) by 1.
5. Return `array`.

### 7.3.17 CreateListFromArrayLike ( `obj` [, `elementTypeList` ] )

The abstract operation `CreateListFromArrayLike` is used to create a `List` value whose elements are provided by the indexed properties of an array-like object, `obj`. The optional argument `elementTypeList` is a `List` containing the names of ECMAScript Language Types that are allowed for element values of the `List` that is created. This abstract operation performs the following steps:

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

### 7.3.18 Invoke ( `V` , `P` [, `argumentsList` ] )

The abstract operation `Invoke` is used to call a method property of an ECMAScript language value. The operation is called with arguments `V`, `P`, and optionally `argumentsList` where `V` serves as both the lookup point for the property and the `this` value of the call, `P` is the property key, and `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. This abstract operation performs the following steps:

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

### 7.3.19 OrdinaryHasInstance ( `C` , `O` )

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

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

### 7.3.20 SpeciesConstructor ( `O, defaultConstructor` )

The abstract operation `SpeciesConstructor` is used to retrieve the constructor that should be used to create new objects that are derived from the argument object `O`. The `defaultConstructor` argument is the constructor to use if a constructor `@@species` property cannot be found starting from `O`. This abstract operation performs the following steps:

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

### 7.3.21 EnumerableOwnProperties ( `O, kind` )

When the abstract operation `EnumerableOwnProperties` is called with `Object O` and `String kind` the following steps are taken:

1. Assert: `Type(O)` is `Object`.
2. Let `ownKeys` be `O.@@ownPropertyKeys`.
3. Let `properties` be a new empty `List`.
4. For each element `key` of `ownKeys` in `List` order, do
   a. If `Type(key)` is `String`, then
      i. Let `desc` be `O.@@ownProperty(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. Order the elements of `properties` so they are in the same relative order as would be produced by the Iterator that would be returned if the `EnumerateObjectProperties` internal method were invoked with `O`.
6. Return `properties`.

### 7.3.22 GetFunctionRealm ( `obj` )

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

1. Assert: `obj` is a callable object.
2. If `obj` has a `[[Realm]]` internal slot, then
   a. Return `obj.[[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 `target` is a non-standard function exotic object that does not have a `[[Realm]]` internal slot.

### 7.4 Operations on Iterator Objects

See Common Iteration Interfaces (25.1).

#### 7.4.1 GetIterator ( `obj` [, `method` ] )
The abstract operation GetIterator with argument \( obj \) and optional argument \( method \) performs the following steps:

1. If \( method \) was not passed, then
   a. Set \( method \) to \(? GetMethod(obj, @@iterator)\).
2. Let \( iterator \) be \(? Call(method, obj)\).
3. If \( Type(iterator) \) is not Object, throw a \( TypeError \) exception.
4. Return \( iterator \).

### 7.4.2 IteratorNext ( \( iterator [ , value ] \) )

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

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

### 7.4.3 IteratorComplete ( \( iterResult \) )

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

1. Assert: \( Type(iterResult) \) is Object.
2. Return \(? ToBoolean(? Get(iterResult, "done")\).

### 7.4.4 IteratorValue ( \( iterResult \) )

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

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

### 7.4.5 IteratorStep ( \( iterator \) )

The abstract operation IteratorStep with argument \( iterator \) requests the next value from \( iterator \) and returns either \( false \) indicating that the iterator has reached its end or the IteratorResult object if a next value is available. IteratorStep performs the following steps:
1. Let \( \text{result} \) be ? \( \text{IteratorNext}(\text{iterator}) \).
2. Let \( \text{done} \) be ? \( \text{IteratorComplete}(\text{result}) \).
3. If \( \text{done} \) is \text{true}, return \text{false}.
4. Return \( \text{result} \).

### 7.4.6 IteratorClose (\text{iterator}, \text{completion})

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

1. Assert: \( \text{Type(\text{iterator}) is Object} \).
2. Assert: \( \text{completion} \) is a Completion Record.
3. Let \( \text{return} \) be ? \( \text{GetMethod(\text{iterator}, \"return\")} \).
4. If \( \text{return} \) is \text{undefined}, return \( \text{Completion(\text{completion})} \).
5. Let \( \text{innerResult} \) be \( \text{Call}(\text{return}, \text{iterator}, \{"\}) \).
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 \text{TypeError} exception.
9. Return \( \text{Completion(\text{completion})} \).

### 7.4.7 CreateIterResultObject (\text{value}, \text{done})

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

1. Assert: \( \text{Type(\text{done}) is Boolean} \).
2. Let \( \text{obj} \) be \( \text{ObjectCreate(%ObjectPrototype%)} \).
3. Perform \( \text{CreateDataProperty(\text{obj}, \"value", \text{value})} \).
4. Perform \( \text{CreateDataProperty(\text{obj}, \"done", \text{done})} \).
5. Return \( \text{obj} \).

### 7.4.8 CreateListIterator (\text{list})

The abstract operation CreateListIterator with argument \text{list} creates an Iterator (25.1.2) object whose next method returns the successive elements of \text{list}. It performs the following steps:

1. Let \( \text{iterator} \) be \( \text{ObjectCreate(%IteratorPrototype%, \"[[IteratedList]], [[ListIteratorNextIndex]]\")} \).
2. Set \( \text{iterator}\.([\text{IteratedList}]) \) to \text{list}.
3. Set \( \text{iterator}\.([\text{ListIteratorNextIndex}]) \) to 0.
4. Let \( \text{next} \) be a new built-in function object as defined in ListIterator \( \text{next} \) (7.4.8.1).
5. Perform \( \text{CreateMethodProperty(\text{iterator}, \"next\", \text{next})} \).
6. Return \textit{iterator}.

\textbf{NOTE} 

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

\subsection*{7.4.8.1 ListIterator \texttt{next}()}

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

1. Let \(O\) be the \texttt{this} value.
2. Assert: \texttt{Type}(\(O\)) is \texttt{Object}.
3. Assert: \(O\) has an [[IteratedList]] internal slot.
4. Let \(list\) be \(O.\text{[[IteratedList]]}\).
5. Let \(index\) be \(O.\text{[[ListIteratorNextIndex]]}\).
6. Let \(len\) be the number of elements of \(list\).
7. If \(index \geq len\), then
   a. Return \texttt{CreateIteratorResultObject(undefined, true)}.
8. Set \(O.\text{[[ListIteratorNextIndex]]}\) to \(index+1\).
9. Return \texttt{CreateIteratorResultObject(list[index], false)}.

\section*{8 Executable Code and Execution Contexts}

\subsection*{8.1 Lexical Environments}

A \textit{Lexical Environment} is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an \textit{Environment Record} and a possibly null reference to an \textit{outer} Lexical Environment. Usually a Lexical Environment is associated with some specific syntactic structure of ECMAScript code such as a \textit{FunctionDeclaration}, a \textit{BlockStatement}, or a \textit{Catch} clause of a \textit{TryStatement} and a new Lexical Environment is created each time such code is evaluated.

An \textit{Environment Record} records the identifier bindings that are created within the scope of its associated Lexical Environment. It is referred to as the Lexical Environment’s \textit{EnvironmentRecord}.

The outer environment reference is used to model the logical nesting of Lexical Environment values. The outer reference of a (inner) Lexical Environment is a reference to the Lexical Environment that logically surrounds the inner Lexical Environment. An outer Lexical Environment may, of course, have its own outer Lexical Environment. A Lexical Environment may serve as the outer environment for multiple inner Lexical Environments. For example, if a \textit{FunctionDeclaration} contains two nested \textit{FunctionDeclarations} then the
Lexical Environments of each of the nested functions will have as their outer Lexical Environment the Lexical Environment of the current evaluation of the surrounding function.

A *global environment* is a Lexical Environment which does not have an outer environment. The *global environment*’s outer environment reference is **null**. A *global environment*’s EnvironmentRecord may be prepopulated with identifier bindings and 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.

A *module environment* is a Lexical Environment that contains the bindings for the top level declarations of a Module. It also contains the bindings that are explicitly imported by the *Module*. The outer environment of a *module environment* is a *global environment*.

A *function environment* is a Lexical Environment that corresponds to the invocation of an ECMAScript function object. A *function environment* may establish a new **this** binding. A *function environment* also captures the state necessary to support **super** method invocations.

Lexical Environments and Environment Record values are purely specification mechanisms and need not correspond to any specific artefact of an ECMAScript implementation. It is impossible for an ECMAScript program to directly access or manipulate such values.

### 8.1.1 Environment Records

There are two primary kinds of Environment Record values used in this specification: *declarative Environment Records* and *object Environment Records*. Declarative Environment Records are used to define the effect of ECMAScript language syntactic elements such as *FunctionDeclarations*, *VariableDeclarations*, and *Catch* clauses that directly associate identifier bindings with ECMAScript language values. Object Environment Records are used to define the effect of ECMAScript elements such as *WithStatement* that associate identifier bindings with the properties of some object. Global Environment Records and function Environment Records are specializations that are used for specifically for *Script* global declarations and for top-level declarations within functions.

For specification purposes Environment Record values are values of the *Record* specification type and 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. The abstract class includes the abstract specification methods defined in Table 14. 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></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>HasBinding(N)</code></td>
<td>Determine if an <em>Environment Record</em> has a binding for the String value <code>N</code>. Return <code>true</code> if it does and <code>false</code> if it does not</td>
</tr>
<tr>
<td><code>CreateMutableBinding(N, D)</code></td>
<td>Create a new but uninitialized mutable binding in an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. If the Boolean argument <code>D</code> is <code>true</code> the binding may be subsequently deleted.</td>
</tr>
<tr>
<td><code>CreateImmutableBinding(N, S)</code></td>
<td>Create a new but uninitialized immutable binding in an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. If <code>S</code> is <code>true</code> 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><code>InitializeBinding(N, V)</code></td>
<td>Set the value of an already existing but uninitialized binding in an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. <code>V</code> is the value for the binding and is a value of any <em>ECMAScript language type</em>.</td>
</tr>
<tr>
<td><code>SetMutableBinding(N, V, S)</code></td>
<td>Set the value of an already existing mutable binding in an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. <code>V</code> is the value for the binding and may be a value of any <em>ECMAScript language type</em>. <code>S</code> is a Boolean flag. If <code>S</code> is <code>true</code> and the binding cannot be set throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td><code>GetBindingValue(N, S)</code></td>
<td>Returns the value of an already existing binding from an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. <code>S</code> is used to identify references originating in <em>strict mode code</em> or that otherwise require strict mode reference semantics. If <code>S</code> is <code>true</code> and the binding does not exist throw a <code>ReferenceError</code> exception. If the binding exists but is uninitialized a <code>ReferenceError</code> is thrown, regardless of the value of <code>S</code>.</td>
</tr>
<tr>
<td><code>DeleteBinding(N)</code></td>
<td>Delete a binding from an <em>Environment Record</em>. The String value <code>N</code> is the text of the bound name. If a binding for <code>N</code> exists, remove the binding and return <code>true</code>. If the binding exists but cannot be removed return <code>false</code>. If the binding does not exist return <code>true</code>.</td>
</tr>
<tr>
<td><code>HasThisBinding()</code></td>
<td>Determine if an <em>Environment Record</em> establishes a <em>this</em> binding. Return <code>true</code> if it does and <code>false</code> if it does not.</td>
</tr>
<tr>
<td><code>HasSuperBinding()</code></td>
<td>Determine if an <em>Environment Record</em> establishes a <em>super</em> method binding. Return <code>true</code> if it does and <code>false</code> if it does not.</td>
</tr>
<tr>
<td><code>WithBaseObject()</code></td>
<td>If this <em>Environment Record</em> is associated with a <em>with</em> statement, return the with object. Otherwise, return <code>undefined</code>.</td>
</tr>
</tbody>
</table>
8.1.1.1 Declarative Environment Records

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

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

8.1.1.1.1 HasBinding (N)

The concrete Environment Record method HasBinding for declarative Environment Records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let envRec be the declarative Environment Record for which the method was invoked.
2. If envRec has a binding for the name that is the value of N, return true.
3. Return false.

8.1.1.1.2 CreateMutableBinding (N, D)

The concrete Environment Record method CreateMutableBinding for declarative Environment Records creates a new mutable binding for the name N that is uninitialized. A binding must not already exist in this Environment Record for N. If Boolean argument D has the value true the new binding is marked as being subject to deletion.

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

8.1.1.1.3 CreateImmutableBinding (N, S)

The concrete Environment Record method CreateImmutableBinding for declarative Environment Records creates a new immutable binding for the name N that is uninitialized. A binding must not already exist in this Environment Record for N. If the Boolean argument S has the value true the new binding is marked as a strict binding.

1. Let envRec be the declarative Environment Record for which the method was invoked.
2. Assert: envRec does not already have a binding for N.
3. Create an immutable binding in envRec for N and record that it is uninitialized. If S is true, record that the newly created binding is a strict binding.
4. Return NormalCompletion(empty).
8.1.1.4 InitializeBinding \((N, V)\)

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

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

8.1.1.5 SetMutableBinding \((N, V, S)\)

The concrete Environment Record method SetMutableBinding for declarative Environment Records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument \(N\) to the value of argument \(V\). A binding for \(N\) normally already exists, but in rare cases it may not. If the binding is an immutable binding, a TypeError is thrown if \(S\) is true.

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

---

**NOTE**

An example of ECMAScript code that results in a missing binding at step 2 is:

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

8.1.1.6 GetBindingValue \((N, S)\)

The concrete Environment Record method GetBindingValue for declarative Environment Records simply returns the value of its bound identifier whose name is the value of the argument \(N\). If the binding exists but is uninitialized a ReferenceError is thrown, regardless of the value of \(S\).
1. Let \texttt{envRec} be the declarative \texttt{Environment Record} for which the method was invoked.
2. Assert: \texttt{envRec} has a binding for \texttt{N}.
3. If the binding for \texttt{N} in \texttt{envRec} is an uninitialized binding, throw a \texttt{ReferenceError} exception.
4. Return the value currently bound to \texttt{N} in \texttt{envRec}.

### 8.1.1.7 DeleteBinding ( \texttt{N} )

The concrete \texttt{Environment Record} method \texttt{DeleteBinding} for declarative Environment Records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let \texttt{envRec} be the declarative \texttt{Environment Record} for which the method was invoked.
2. Assert: \texttt{envRec} has a binding for the name that is the value of \texttt{N}.
3. If the binding for \texttt{N} in \texttt{envRec} cannot be deleted, return \texttt{false}.
4. Remove the binding for \texttt{N} from \texttt{envRec}.
5. Return \texttt{true}.

### 8.1.1.8 HasThisBinding ()

Regular declarative Environment Records do not provide a \texttt{this} binding.

1. Return \texttt{false}.

### 8.1.1.9 HasSuperBinding ()

Regular declarative Environment Records do not provide a \texttt{super} binding.

1. Return \texttt{false}.

### 8.1.1.10 WithBaseObject ()

Declarative Environment Records always return \texttt{undefined} as their \texttt{WithBaseObject}.

1. Return \texttt{undefined}.

### 8.1.2 Object Environment Records

Each object \texttt{Environment Record} is associated with an object called its \textit{binding object}. An object \texttt{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 \texttt{Environment Record} may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a
result of such a side-effect are considered to be a mutable binding even if the Writable attribute of the corresponding property has the value **false**. Immutable bindings do not exist for object Environment Records.

Object Environment Records created for **with** statements (13.11) can provide their binding object as an implicit **this** value for use in function calls. The capability is controlled by a **withEnvironment** Boolean value that is associated with each object **Environment Record**. By default, the value of **withEnvironment** is **false** for any object **Environment Record**.

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

### 8.1.1.2.1 HasBinding ( \( N \) )

The concrete **Environment Record** method HasBinding for object Environment Records determines if its associated binding object has a property whose name is the value of the argument \( N \):

1. Let \( envRec \) be the object **Environment Record** for which the method was invoked.
2. Let \( bindings \) be the binding object for \( envRec \).
3. Let \( foundBinding \) be \( ? \) HasProperty(\( bindings \), \( N \)).
4. If \( foundBinding \) is **false**, return **false**.
5. If the **withEnvironment** flag of \( envRec \) is **false**, return **true**.
6. Let \( unscopables \) be \( ? \) Get(\( bindings \), @@unscopables).
7. If Type(\( unscopables \)) is Object, then
   a. Let \( blocked \) be ToBoolean(\( ? \) Get(\( unscopables \), \( N \))).
   b. If \( blocked \) is **true**, return **false**.
8. Return **true**.

### 8.1.1.2.2 CreateMutableBinding ( \( N, D \) )

The concrete **Environment Record** method CreateMutableBinding for object Environment Records creates in an **Environment Record**'s associated binding object a property whose name is the String value and initializes it to the value **undefined**. If Boolean argument \( D \) has the value **true** the new property’s [[Configurable]] attribute is set to **true**; otherwise it is set to **false**.

1. Let \( envRec \) be the object **Environment Record** for which the method was invoked.
2. Let \( bindings \) be the binding object for \( envRec \).
3. 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.
8.1.1.2.3 CreateImmutableBinding \((N, S)\)

The concrete `Environment Record` method CreateImmutableBinding is never used within this specification in association with object `Environment Records`.

8.1.1.2.4 InitializeBinding \((N, V)\)

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

1. Let `envRec` be the object `Environment Record` for which the method was invoked.
2. Assert: `envRec` must have an uninitialized binding for \(N\).
3. Record that the binding for \(N\) in `envRec` has been initialized.

**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, implementations do not need to explicitly track the initialization state of individual object `Environment Record` bindings.

8.1.1.2.5 SetMutableBinding \((N, V, S)\)

The concrete `Environment Record` method SetMutableBinding for object `Environment Records` attempts to set the value of the `Environment Record`'s associated binding object's property whose name is the value of the argument \(N\) to the value of argument \(V\). A property named \(N\) normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument \(S\).

1. Let `envRec` be the object `Environment Record` for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.

8.1.1.2.6 GetBindingValue \((N, S)\)

The concrete `Environment Record` method GetBindingValue for object `Environment Records` returns the value of its associated binding object's property whose name is the String value of the argument identifier \(N\). The property should already exist but if it does not the result depends upon the value of the \(S\) argument:

1. Let `envRec` be the object `Environment Record` for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.
3. Let `value` be ? `HasProperty(bindings, N)`.
4. If `value` is `false`, then
   a. If \(S\) is `false`, return the value `undefined`; otherwise throw a `ReferenceError` exception.
Return `bindings, N`.

### 8.1.1.2.7 DeleteBinding (N)

The concrete `Environment Record` method `DeleteBinding` for object `Environment Records` can only delete bindings that correspond to properties of the environment object whose `[[Configurable]]` attribute have the value `true`.

1. Let `envRec` be the object `Environment Record` for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.
3. Return `bindings.[[Delete]](N)`.

### 8.1.1.2.8 HasThisBinding ()

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

1. Return `false`.

### 8.1.1.2.9 HasSuperBinding ()

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

1. Return `false`.

### 8.1.1.2.10 WithBaseObject ()

Object `Environment Records` return `undefined` as their `WithBaseObject` unless their `withEnvironment` flag is `true`.

1. Let `envRec` be the object `Environment Record` for which the method was invoked.
2. If the `withEnvironment` flag of `envRec` is `true`, return the binding object for `envRec`.
3. Otherwise, return `undefined`.

### 8.1.1.3 Function Environment Records

A `function Environment Record` is a declarative `Environment Record` that is used to represent the 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 15.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>Value</td>
<td>Meaning</td>
</tr>
</tbody>
</table>

The `Function Environment Records` have the additional state fields listed in Table 15.
Function Environment Records support all of the declarative `Environment Record` methods listed in Table 14 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 16:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BindThisValue(V)</td>
<td>Set the <code>[[ThisValue]]</code> and record that it has been initialized.</td>
</tr>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this <code>Environment Record</code>'s <code>this</code> binding. Throws a <code>ReferenceError</code> if the <code>this</code> binding has not been initialized.</td>
</tr>
<tr>
<td>GetSuperBase()</td>
<td>Return the object that is the base for <code>super</code> property accesses bound in this <code>Environment Record</code>. The object is derived from this <code>Environment Record</code>'s <code>[[HomeObject]]</code> field. The value <code>undefined</code> indicates that <code>super</code> 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:
8.1.1.3.1 BindThisValue (V)

1. Let envRec be the function Environment Record for which the method was invoked.
2. Assert: envRec.[[ThisBindingStatus]] is not "lexical".
3. If envRec.[[ThisBindingStatus]] is "initialized", throw a ReferenceError exception.
4. Set envRec.[[ThisValue]] to V.
5. Set envRec.[[ThisBindingStatus]] to "initialized".
6. Return V.

8.1.1.3.2 HasThisBinding ()

1. Let envRec be the function Environment Record for which the method was invoked.
2. If envRec.[[ThisBindingStatus]] is "lexical", return false; otherwise, return true.

8.1.1.3.3 HasSuperBinding ()

1. Let envRec be the function Environment Record for which the method was invoked.
2. If envRec.[[ThisBindingStatus]] is "lexical", return false.
3. If envRec.[[HomeObject]] has the value undefined, return false; otherwise, return true.

8.1.1.3.4 GetThisBinding ()

1. Let envRec be the function Environment Record for which the method was invoked.
2. Assert: envRec.[[ThisBindingStatus]] is not "lexical".
3. If envRec.[[ThisBindingStatus]] is "uninitialized", throw a ReferenceError exception.
4. Return envRec.[[ThisValue]].

8.1.1.3.5 GetSuperBase ()

1. Let envRec be the function Environment Record for which the method was invoked.
2. Let home be envRec.[[HomeObject]].
3. If home has the value undefined, return undefined.
4. Assert: Type(home) is Object.
5. Return ? home.[[GetPrototypeOf]]().

8.1.1.4 Global Environment Records

A global Environment Record is used to represent the outer most scope that is shared by all of the ECMAScript Script elements that are processed in a common realm. A global Environment Record provides the bindings for built-in globals (clause 18), properties of the global object, and for all top-level declarations (13.2.8, 13.2.10) 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 18) and all bindings introduced by a **FunctionDeclaration**, **GeneratorDeclaration**, **AsyncFunctionDeclaration** 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**, 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 17** and the additional methods listed in **Table 18**.

### Table 17: Additional Fields of Global Environment Records

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

### Table 18: Additional Methods of Global Environment Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this Environment Record’s this binding.</td>
</tr>
<tr>
<td>HasVarDeclaration (N)</td>
<td>Determines if the argument identifier has a binding in this Environment Record that was created using a VariableDeclaration, FunctionDeclaration, GeneratorDeclaration, or AsyncFunctionDeclaration.</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 lexically 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 undefined a global var binding in the [[ObjectRecord]] component of a global Environment Record. The binding will be a mutable binding. The corresponding global object property will have attribute values appropriate for a var. The String value 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 function binding in the [[ObjectRecord]] component of a global Environment Record. The binding will be a mutable binding. The corresponding global object property will have attribute values appropriate for a function. The String value 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.
8.1.4.1 HasBinding (N)

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

1. Let envRec be the global Environment Record for which the method was invoked.
2. Let DclRec be envRec.[[DeclarativeRecord]].
3. If DclRec.HasBinding(N) is true, return true.
4. Let ObjRec be envRec.[[ObjectRecord]].

8.1.4.2 CreateMutableBinding (N, D)

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

1. Let envRec be the global Environment Record for which the method was invoked.
2. Let DclRec be envRec.[[DeclarativeRecord]].
3. If DclRec.HasBinding(N) is true, throw a TypeError exception.

8.1.4.3 CreateImmutableBinding (N, S)

The concrete Environment Record method CreateImmutableBinding for global Environment Records creates a new immutable binding for the name N that is uninitialized. A binding must not already exist in this Environment Record for N. If the Boolean argument S has the value true the new binding is marked as a strict binding.

1. Let envRec be the global Environment Record for which the method was invoked.
2. Let DclRec be envRec.[[DeclarativeRecord]].
3. If DclRec.HasBinding(N) is true, throw a TypeError exception.

8.1.4.4 InitializeBinding (N, V)

The concrete Environment Record method InitializeBinding for global Environment Records is used to set the bound value of the current binding of the identifier whose name is the value of the argument N to the value of argument V. An uninitialized binding for N must already exist.

1. Let envRec be the global Environment Record for which the method was invoked.
2. Let DclRec be envRec.[[DeclarativeRecord]].
3. If \textit{DclRec}.\text{HasBinding}(N) is \textbf{true}, then
4. Assert: If the binding exists, it must be in the object \textit{Environment Record}.
5. Let \textit{ObjRec} be envRec.\[\text{ObjectRecord}\].

\textbf{8.1.1.4.5 SetMutableBinding (N, V, S)}

The concrete \textit{Environment Record} method \textit{SetMutableBinding} for global \textit{Environment Records} attempts to change 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}. If the binding is an immutable binding, a \textbf{TypeError} is thrown if \textit{S} is \textbf{true}. A property named \textit{N} normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument \textit{S}.

1. Let \textit{envRec} be the global \textit{Environment Record} for which the method was invoked.
2. Let \textit{DclRec} be envRec.\[\text{DeclarativeRecord}\].
3. If \textit{DclRec}.\text{HasBinding}(N) is \textbf{true}, then
4. Let \textit{ObjRec} be envRec.\[\text{ObjectRecord}\].

\textbf{8.1.1.4.6 GetBindingValue (N, S)}

The concrete \textit{Environment Record} method \textit{GetBindingValue} for global \textit{Environment Records} returns the value of its bound identifier whose name is the value of the argument \textit{N}. If the binding is an uninitialized binding throw a \textbf{ReferenceError} exception. A property named \textit{N} normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument \textit{S}.

1. Let \textit{envRec} be the global \textit{Environment Record} for which the method was invoked.
2. Let \textit{DclRec} be envRec.\[\text{DeclarativeRecord}\].
3. If \textit{DclRec}.\text{HasBinding}(N) is \textbf{true}, then
4. Let \textit{ObjRec} be envRec.\[\text{ObjectRecord}\].

\textbf{8.1.1.4.7 DeleteBinding (N)}

The concrete \textit{Environment Record} method \textit{DeleteBinding} for global \textit{Environment Records} can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let \textit{envRec} be the global \textit{Environment Record} for which the method was invoked.
2. Let \textit{DclRec} be envRec.\[\text{DeclarativeRecord}\].
3. If \( DclRec.\text{HasBinding}(N) \) is \textbf{true}, then
   a. Return \( DclRec.\text{DeleteBinding}(N) \).
4. Let \( ObjRec \) be \( envRec.[[\text{ObjectRecord}]] \).
5. Let \( \text{globalObject} \) be the binding object for \( ObjRec \).
6. Let \( \text{existingProp} \) be \( \text{HasOwnProperty}(\text{globalObject}, N) \).
7. If \( \text{existingProp} \) is \textbf{true}, then
   a. Let \( \text{status} \) be \( ObjRec.\text{DeleteBinding}(N) \).
   b. If \( \text{status} \) is \textbf{true}, then
      i. Let \( \text{varNames} \) be \( envRec.[[\text{VarNames}]] \).
      ii. If \( N \) is an element of \( \text{varNames} \), remove that element from the \( \text{varNames} \).
   c. Return \( \text{status} \).
8. Return \textbf{true}.

\textbf{8.1.1.4.8 \hspace{1em} HasThisBinding ( )}

1. Return \textbf{true}.

\textbf{8.1.1.4.9 \hspace{1em} HasSuperBinding ( )}

1. Return \textbf{false}.

\textbf{8.1.1.4.10 \hspace{1em} WithBaseObject ( )}

Global Environment Records always return \textbf{undefined} as their WithBaseObject.

1. Return \textbf{undefined}.

\textbf{8.1.1.4.11 \hspace{1em} GetThisBinding ( )}

1. Let \( envRec \) be the global \textit{Environment Record} for which the method was invoked.
2. Return \( envRec.[[\text{GlobalThisValue}]] \).

\textbf{8.1.1.4.12 \hspace{1em} HasVarDeclaration ( \( N \) )}

The concrete \textit{Environment Record} method \textit{HasVarDeclaration} for global Environment Records determines if the argument identifier has a binding in this record that was created using a \textit{VariableStatement} or a \textit{FunctionDeclaration}:

1. Let \( envRec \) be the global \textit{Environment Record} for which the method was invoked.
2. Let \( \text{varDeclaredNames} \) be \( envRec.[[\text{VarNames}]] \).
3. If \( \text{varDeclaredNames} \) contains \( N \), return \textbf{true}.
4. Return \textbf{false}.
8.1.1.4.13 HasLexicalDeclaration \( (N) \)

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

1. Let \( \text{envRec} \) be the global Environment Record for which the method was invoked.
2. Let \( \text{DclRec} \) be \( \text{envRec}[[\text{DeclarativeRecord}]] \).
3. Return \( \text{DclRec}.\text{HasBinding}(N) \).

8.1.1.4.14 HasRestrictedGlobalProperty \( (N) \)

The concrete Environment Record method HasRestrictedGlobalProperty for global Environment Records determines if the argument identifier is the name of a property of the global object that must not be shadowed by a global lexically binding:

1. Let \( \text{envRec} \) be the global Environment Record for which the method was invoked.
2. Let \( \text{ObjRec} \) be \( \text{envRec}[[\text{ObjectRecord}]] \).
3. Let \( \text{globalObject} \) be the binding object for \( \text{ObjRec} \).
4. Let \( \text{existingProp} \) be \( ? \text{globalObject}[[\text{GetOwnProperty}]][N] \).
5. If \( \text{existingProp} \) is undefined, return false.
6. If \( \text{existingProp}.[[\text{Configurable}]] \) is true, return false.
7. 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.

8.1.1.4.15 CanDeclareGlobalVar \( (N) \)

The concrete Environment Record method CanDeclareGlobalVar for global Environment Records determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument \( N \). Redundant var declarations and var declarations for pre-existing global object properties are allowed.

1. Let \( \text{envRec} \) be the global Environment Record for which the method was invoked.
2. Let \( \text{ObjRec} \) be \( \text{envRec}[[\text{ObjectRecord}]] \).
3. Let \( \text{globalObject} \) be the binding object for \( \text{ObjRec} \).
4. Let \( \text{hasProperty} \) be \( ? \text{HasOwnProperty}(\text{globalObject}, N) \).
5. If \( \text{hasProperty} \) is true, return true.
6. Return ? IsExtensible(\( \text{globalObject} \)).

8.1.1.4.16 CanDeclareGlobalFunction \( (N) \)
The concrete *Environment Record* method CanDeclareGlobalFunction for global Environment Records determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument \( N \).

1. Let \( \text{envRec} \) be the global *Environment Record* for which the method was invoked.
2. Let \( \text{ObjRec} \) be \( \text{envRec}.[[\text{ObjectRecord}]] \).
3. Let \( \text{globalObject} \) be the binding object for \( \text{ObjRec} \).
4. Let \( \text{existingProp} \) be \( ? \text{globalObject}.[[\text{GetOwnProperty}]](N) \).
5. If \( \text{existingProp} \) is \( \text{undefined} \), return \( ? \text{IsExtensible}(\text{globalObject}) \).
6. If \( \text{existingProp}.[[\text{Configurable}]] \) is \( \text{true} \), return \( \text{true} \).
7. If \( \text{IsDataDescriptor}(\text{existingProp}) \) is \( \text{true} \) and \( \text{existingProp} \) has attribute values \{[[\text{Writable}]]: \text{true},

[[\text{Enumerable}]]: \text{true}\}, return \( \text{true} \).
8. Return \( \text{false} \).

### 8.1.1.4.17 CreateGlobalVarBinding ( \( N, D \) )

The concrete *Environment Record* method CreateGlobalVarBinding for global Environment Records creates and initializes a mutable binding in the associated object *Environment Record* and records the bound name in the associated \([\langle \text{VarNames} \rangle \] List. If a binding already exists, it is reused and assumed to be initialized.

1. Let \( \text{envRec} \) be the global *Environment Record* for which the method was invoked.
2. Let \( \text{ObjRec} \) be \( \text{envRec}.[[\text{ObjectRecord}]] \).
3. Let \( \text{globalObject} \) be the binding object for \( \text{ObjRec} \).
4. Let \( \text{hasProperty} \) be \( ? \text{HasOwnProperty}(\text{globalObject}, N) \).
5. Let \( \text{extensible} \) be \( ? \text{IsExtensible}(\text{globalObject}) \).
6. If \( \text{hasProperty} \) is \( \text{false} \) and \( \text{extensible} \) is \( \text{true} \), then
   a. Perform \( ? \text{ObjRec}.\text{CreateMutableBinding}(N, D) \).
   b. Perform \( ? \text{ObjRec}.\text{InitializeBinding}(N, \text{undefined}) \).
7. Let \( \text{varDeclaredNames} \) be \( \text{envRec}.[[\text{VarNames}]] \).
8. If \( \text{varDeclaredNames} \) does not contain \( N \), then
   a. Append \( N \) to \( \text{varDeclaredNames} \).
9. Return \( \text{NormalCompletion} \) (empty).

### 8.1.1.4.18 CreateGlobalFunctionBinding ( \( N, V, D \) )

The concrete *Environment Record* method CreateGlobalFunctionBinding for global Environment Records creates and initializes a mutable binding in the associated object *Environment Record* and records the bound name in the associated \([\langle \text{VarNames} \rangle \] List. If a binding already exists, it is replaced.

1. Let \( \text{envRec} \) be the global *Environment Record* for which the method was invoked.
2. Let \( \text{ObjRec} \) be \( \text{envRec}.[[\text{ObjectRecord}]] \).
3. Let `globalObject` be the binding object for `ObjRec`.
4. Let `existingProp` be `globalObject. [[GetOwnProperty]](N)`.
5. If `existingProp` is `undefined` or `existingProp. [[Configurable]]` is `true`, then
   a. Let `desc` be the `PropertyDescriptor`{`[[Value]]`: `V`, `[[Writable]]`: `true`, `[[Enumerable]]`: `true`, `[[Configurable]]`: `D`}.
6. Else,
   a. Let `desc` be the `PropertyDescriptor`{`[[Value]]`: `V`}.
8. Record that the binding for `N` in `ObjRec` has been initialized.
10. Let `varDeclaredNames` be `envRec. [[VarNames]]`.
11. If `varDeclaredNames` does not contain `N`, then
    a. Append `N` to `varDeclaredNames`.
12. Return `NormalCompletion(empty)`.

**NOTE** Global function declarations are always represented as own properties of the `global object`. If possible, an existing own property is reconfigured to have a standard set of attribute values. Steps 8-9 are equivalent to what calling the InitializeBinding concrete method would do and if `globalObject` is a Proxy will produce the same sequence of Proxy trap calls.

### 8.1.1.5 Module Environment Records

A module `Environment Record` is a declarative `Environment Record` that is used to represent the outer scope of an ECMAScript `Module`. In addition to normal mutable and immutable bindings, module Environment Records also provide immutable import bindings which are bindings that provide indirect access to a target binding that exists in another `Environment Record`.

Module Environment Records support all of the declarative `Environment Record` methods listed in Table 14 and share the same specifications for all of those methods except for `GetBindingValue`, `DeleteBinding`, `HasThisBinding` and `GetThisBinding`. In addition, module Environment Records support the methods listed in Table 19:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CreateImportBinding(N, M, N2)</code></td>
<td>Create an immutable indirect binding in a module <code>Environment Record</code>. The String value <code>N</code> is the text of the bound name. <code>M</code> is a <code>Module Record</code>, and <code>N2</code> is a binding that exists in M’s module <code>Environment Record</code>.</td>
</tr>
<tr>
<td><code>GetThisBinding()</code></td>
<td>Return the value of this <code>Environment Record</code>'s <code>this</code> binding.</td>
</tr>
</tbody>
</table>
The behaviour of the additional concrete specification methods for module Environment Records are defined by the following algorithms:

### 8.1.1.5.1 GetBindingValue \((N, S)\)

The concrete Environment Record method GetBindingValue for module Environment Records 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.

1. Assert: \(S\) is true.
2. Let \(envRec\) be the module Environment Record for which the method was invoked.
3. Assert: \(envRec\) has a binding for \(N\).
4. 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 \(M[[\text{Environment}]]\).
   c. If targetEnv is undefined, throw a ReferenceError exception.
   d. Let targetER be targetEnv's EnvironmentRecord.
   e. Return \(? targetER.GetBindingValue(N2, true)\).
5. If the binding for \(N\) in \(envRec\) is an uninitialized binding, throw a ReferenceError exception.
6. Return the value currently bound to \(N\) in \(envRec\).

**NOTE** \(S\) will always be true because a Module is always strict mode code.

### 8.1.1.5.2 DeleteBinding \((N)\)

The concrete Environment Record method DeleteBinding for module Environment Records refuses to delete bindings.

1. Assert: This method is never invoked. See 12.5.3.1.

**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 that would resolve to a module Environment Record binding. See 12.5.3.1.

### 8.1.1.5.3 HasThisBinding \()

Module Environment Records provide a this binding.

1. Return true.

### 8.1.1.5.4 GetThisBinding \()
1. Return `undefined`.

### 8.1.1.5.5 CreateImportBinding \((N, M, N2)\)

The concrete Environment Record method CreateImportBinding for module Environment Records creates a new initialized immutable indirect binding for the name \(N\). A binding must not already exist in this Environment Record for \(N\). \(M\) is a Module Record, and \(N2\) 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.

1. Let `envRec` be the module Environment Record for which the method was invoked.
2. Assert: `envRec` does not already have a binding for \(N\).
3. Assert: \(M\) is a Module Record.
4. Assert: When \(M[[\text{Environment}]]\) is instantiated it will have a direct binding for \(N2\).
5. 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.
6. Return `NormalCompletion(empty)`.

### 8.1.2 Lexical Environment Operations

The following abstract operations are used in this specification to operate upon lexical environments:

#### 8.1.2.1 GetIdentifierReference \((lex, name, strict)\)

The abstract operation GetIdentifierReference is called with a Lexical Environment `lex`, a String `name`, and a Boolean flag `strict`. The value of `lex` may be `null`. When called, the following steps are performed:

1. If `lex` is the value `null`, then
   a. Return a value of type `Reference` whose base value component is `undefined`, whose referenced name component is `name`, and whose strict reference flag is `strict`.
2. Let `envRec` be `lex`'s EnvironmentRecord.
3. Let `exists` be `? envRec.HasBinding(name)`.
4. If `exists` is `true`, then
   a. Return a value of type `Reference` whose base value component is `envRec`, whose referenced name component is `name`, and whose strict reference flag is `strict`.
5. Else,
   a. Let `outer` be the value of `lex`'s outer environment reference.
   b. Return `? GetIdentifierReference(outer, name, strict)`.

#### 8.1.2.2 NewDeclarativeEnvironment \((E)\)
When the abstract operation NewDeclarativeEnvironment is called with a Lexical Environment as argument $E$ the following steps are performed:

1. Let $env$ be a new Lexical Environment.
2. Let $envRec$ be a new declarative Environment Record containing no bindings.
3. Set $env$'s EnvironmentRecord to $envRec$.
4. Set the outer lexical environment reference of $env$ to $E$.
5. Return $env$.

### 8.1.2.3 NewObjectEnvironment ( $O, E$ )

When the abstract operation NewObjectEnvironment is called with an Object $O$ and a Lexical Environment $E$ as arguments, the following steps are performed:

1. Let $env$ be a new Lexical Environment.
2. Let $envRec$ be a new object Environment Record containing $O$ as the binding object.
3. Set $env$'s EnvironmentRecord to $envRec$.
4. Set the outer lexical environment reference of $env$ to $E$.
5. Return $env$.

### 8.1.2.4 NewFunctionEnvironment ( $F, newTarget$ )

When the abstract operation NewFunctionEnvironment is called with arguments $F$ and $newTarget$ the following steps are performed:

1. Assert: $F$ is an ECMAScript function.
2. Assert: Type($newTarget$) is Undefined or Object.
3. Let $env$ be a new Lexical Environment.
4. Let $envRec$ be a new function Environment Record containing no bindings.
5. Set $envRec$.[[FunctionObject]] to $F$.
6. If $F$.[[ThisMode]] is lexical, set $envRec$.[[ThisBindingStatus]] to "lexical".
7. Else, set $envRec$.[[ThisBindingStatus]] to "uninitialized".
8. Let $home$ be $F$.[[HomeObject]].
9. Set $envRec$.[[HomeObject]] to $home$.
10. Set $envRec$.[[NewTarget]] to $newTarget$.
11. Set $env$'s EnvironmentRecord to $envRec$.
12. Set the outer lexical environment reference of $env$ to $F$.[[Environment]].

### 8.1.2.5 NewGlobalEnvironment ( $G, thisValue$ )
When the abstract operation NewGlobalEnvironment is called with arguments \( G \) and \( thisValue \), the following steps are performed:

1. Let \( env \) be a new \textit{Lexical Environment}.
2. Let \( objRec \) be a new object \textit{Environment Record} containing \( G \) as the binding object.
3. Let \( dclRec \) be a new declarative \textit{Environment Record} containing no bindings.
4. Let \( globalRec \) be a new global \textit{Environment Record}.
5. Set \( globalRec.\text{[[ObjectRecord]]} \) to \( objRec \).
6. Set \( globalRec.\text{[[GlobalThisValue]]} \) to \( thisValue \).
7. Set \( globalRec.\text{[[DeclarativeRecord]]} \) to \( dclRec \).
8. Set \( globalRec.\text{[[VarNames]]} \) to a new empty \textit{List}.
9. Set \( env \)'s \textit{EnvironmentRecord} to \( globalRec \).
10. Set the outer lexical environment reference of \( env \) to \textit{null}.
11. Return \( env \).

### 8.1.2.6 NewModuleEnvironment ( \( E \) )

When the abstract operation NewModuleEnvironment is called with a \textit{Lexical Environment} argument \( E \) the following steps are performed:

1. Let \( env \) be a new \textit{Lexical Environment}.
2. Let \( envRec \) be a new module \textit{Environment Record} containing no bindings.
3. Set \( env \)'s \textit{EnvironmentRecord} to \( envRec \).
4. Set the outer lexical environment reference of \( env \) to \( E \).
5. Return \( env \).

### 8.2 Realms

Before it is evaluated, all ECMAScript code must be associated with a \textit{realm}. Conceptually, a \textit{realm} consists of a set of intrinsic objects, an ECMAScript \textit{global environment}, all of the ECMAScript code that is loaded within the scope of that \textit{global environment}, and other associated state and resources.

A \textit{realm} is represented in this specification as a \textit{Realm Record} with the fields specified in \textit{Table 20}:

**Table 20: Realm Record Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Intrinsics]]</td>
<td>\textit{Record} whose field names are intrinsic keys and</td>
<td>The intrinsic values used by code associated with this \textit{realm}</td>
</tr>
<tr>
<td>[[GlobalObject]]</td>
<td>Object</td>
<td>The global object for this realm</td>
</tr>
<tr>
<td>[[GlobalEnv]]</td>
<td>Lexical Environment</td>
<td>The global environment for this realm</td>
</tr>
<tr>
<td>[[TemplateMap]]</td>
<td>A List of Record { [Strings]: List, [Array]: Object}.</td>
<td>Template objects are canonicalized separately for each realm using its Realm Record's [[TemplateMap]]. Each [[Strings]] value is a List containing, in source text order, the raw String values of a TemplateLiteral that has been evaluated. The associated [[Array]] value is the corresponding template object that is passed to a tag function.</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is undefined.</td>
<td>Field reserved for use by host environments that need to associate additional information with a Realm Record.</td>
</tr>
</tbody>
</table>

### 8.2.1 CreateRealm ( )

The abstract operation CreateRealm with no arguments performs the following steps:

1. Let `realmRec` be a new Realm Record.
2. Perform CreateIntrinsics(`realmRec`).
3. Set `realmRec`.[[GlobalObject]] to `undefined`.
4. Set `realmRec`.[[GlobalEnv]] to `undefined`.
5. Set `realmRec`.[[TemplateMap]] to a new empty List.
6. Return `realmRec`.

### 8.2.2 CreateIntrinsics ( `realmRec` )

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

1. Let `intrinsics` be a new Record.
2. Set `realmRec`.[[Intrinsics]] to `intrinsics`.
3. Let `objProto` be `ObjectCreate(null)`.
4. Set `intrinsics`.[%ObjectPrototype%] to `objProto`.
5. Let `throwerSteps` be the algorithm steps specified in 9.2.7.1 for the %ThrowTypeError% function.
6. Let `thrower` be `CreateBuiltInFunction(realmRec, throwerSteps, null)`.
7. Set `intrinsics`.[%ThrowTypeError%] to `thrower`.
8. Let \( \text{noSteps} \) be an empty sequence of algorithm steps.
9. Let \( \text{funcProto} \) be \( \text{CreateBuiltInFunction}(\text{realmRec}, \text{noSteps}, \text{objProto}) \).
10. Set \( \text{intrinsic} \).[[%FunctionPrototype%]] to \( \text{funcProto} \).
11. Call \( \text{thrower}.[[\text{SetPrototypeOf}]](\text{funcProto}) \).
12. Perform \( \text{AddRestrictedFunctionProperties}(\text{funcProto}, \text{realmRec}) \).
13. Set fields of \( \text{intrinsic} \) with the values listed in Table 7 that have not already been handled above. The field names are the names listed in column one of the table. The value of each field is a new object value fully and recursively populated with property values as defined by the specification of each object in clauses 18-26. All object property values are newly created object values. All values that are built-in function objects are created by performing \( \text{CreateBuiltInFunction}(\text{realmRec}, <\text{steps}>, <\text{prototype}>, <\text{slots}>) \) where \( <\text{steps}> \) is the definition of that function provided by this specification, \( <\text{prototype}> \) is the specified value of the function's [[Prototype]] internal slot and \( <\text{slots}> \) is a list of the names, if any, of the function's specified internal slots. The creation of the intrinsic and their properties must be ordered to avoid any dependencies upon objects that have not yet been created.
14. Return \( \text{intrinsic} \).

### 8.2.3 SetRealmGlobalObject ( \( \text{realmRec}, \text{globalObj}, \text{thisValue} \) )

The abstract operation SetRealmGlobalObject with arguments \( \text{realmRec}, \text{globalObj} \), and \( \text{thisValue} \) performs the following steps:

1. If \( \text{globalObj} \) is \( \text{undefined} \), then
   a. Let \( \text{intrinsic} \) be \( \text{realmRec}.[[\text{Intrinsic}]] \).
   b. Set \( \text{globalObj} \) to \( \text{ObjectCreate}(\text{intrinsic}.[[\text{ObjectPrototype}%]]) \).
2. Assert: \( \text{Type}(\text{globalObj}) \) is Object.
3. If \( \text{thisValue} \) is \( \text{undefined} \), set \( \text{thisValue} \) to \( \text{globalObj} \).
4. Set \( \text{realmRec}.[[\text{GlobalObject}]] \) to \( \text{globalObj} \).
5. Let \( \text{newGlobalEnv} \) be \( \text{NewGlobalEnvironment}(\text{globalObj}, \text{thisValue}) \).
6. Set \( \text{realmRec}.[[\text{GlobalEnv}]] \) to \( \text{newGlobalEnv} \).
7. Return \( \text{realmRec} \).

### 8.2.4 SetDefaultGlobalBindings ( \( \text{realmRec} \) )

The abstract operation SetDefaultGlobalBindings with argument \( \text{realmRec} \) performs the following steps:

1. Let \( \text{global} \) be \( \text{realmRec}.[[\text{GlobalObject}]] \).
2. For each property of the Global Object specified in clause 18, do
   a. Let \( \text{name} \) be the String value of the property name.
   b. Let \( \text{desc} \) be the fully populated data property descriptor for the property containing the specified attributes for the property. For properties listed in 18.2, 18.3, or 18.4 the value of the [[Value]] attribute is the corresponding intrinsic object from \( \text{realmRec} \).
Perform \texttt{DefinePropertyOrThrow}(\textit{global}, \textit{name}, \textit{desc}).

3. Return \textit{global}.

### 8.3 Execution Contexts

An \textit{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 \textit{execution context} per \textit{agent} that is actually executing code. This is known as the \textit{agent's running execution context}. All references to the \textit{running execution context} in this specification denote the \textit{running execution context} of the surrounding \textit{agent}.

The \textit{execution context stack} is used to track execution contexts. The \textit{running execution context} is always the top element of this stack. A new \textit{execution context} is created whenever control is transferred from the executable code associated with the currently \textit{running execution context} to executable code that is not associated with that execution context. The newly created \textit{execution context} is pushed onto the stack and becomes the \textit{running execution context}.

An \textit{execution context} contains whatever implementation specific state is necessary to track the execution progress of its associated code. Each \textit{execution context} has at least the state components listed in Table 21.

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

Evaluation of code by the \textit{running execution context} may be suspended at various points defined within this specification. Once the \textit{running execution context} has been suspended a different \textit{execution context} may become the \textit{running execution context} and commence evaluating its code. At some later time a suspended \textit{execution context} may again become the \textit{running execution context} and continue evaluating its code at the point where it had previously been suspended. Transition of the \textit{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 22.

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

The LexicalEnvironment and VariableEnvironment components of an execution context are always Lexical Environments. When an execution context is created its LexicalEnvironment and VariableEnvironment components initially have the same value.

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

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

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

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation. It is impossible for ECMAScript code to directly access or observe an execution context.

8.3.1 GetActiveScriptOrModule ()
The GetActiveScriptOrModule abstract operation is used to determine the running script or module, based on the running execution context. GetActiveScriptOrModule performs the following steps:

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 component.

### 8.3.2 ResolveBinding (\( name \ [, \ env \ ] \))

The ResolveBinding abstract operation is used to determine the binding of \( name \) passed as a String value. The optional argument \( env \) can be used to explicitly provide the Lexical Environment that is to be searched for the binding. During execution of ECMAScript code, ResolveBinding is performed using the following algorithm:

1. If \( env \) was not passed or if \( env \) is undefined, then
   a. Set \( env \) to the running execution context’s LexicalEnvironment.
2. Assert: \( env \) is a Lexical Environment.
3. If the code matching the syntactic production that is being evaluated is contained in strict mode code, let \( strict \) be true, else let \( strict \) be false.
4. Return ? GetIdentifierReference\((env, name, strict)\).

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

### 8.3.3 GetThisEnvironment ( )

The abstract operation GetThisEnvironment finds the Environment Record that currently supplies the binding of the keyword this. GetThisEnvironment performs the following steps:

1. Let \( lex \) be the running execution context’s LexicalEnvironment.
2. Repeat,
   a. Let \( envRec \) be \( lex \)'s EnvironmentRecord.
   b. Let \( exists \) be \( envRec\).HasThisBinding().
   c. If \( exists \) is true, return \( envRec \).
   d. Let \( outer \) be the value of \( lex \)'s outer environment reference.
   e. Assert: \( outer \) is not null.
   f. Set \( lex \) to \( outer \).

**NOTE** The loop in step 2 will always terminate because the list of environments always ends with the global environment which has a this binding.
8.3.4 ResolveThisBinding ()

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

1. Let `envRec` be `GetThisEnvironment()`. 
2. Return `? envRec.GetThisBinding()`.

8.3.5 GetNewTarget ()

The abstract operation GetNewTarget determines the NewTarget value using the LexicalEnvironment of the running execution context. GetNewTarget performs the following steps:

1. Let `envRec` be `GetThisEnvironment()`. 
2. Assert: `envRec` has a `[[NewTarget]]` field.

8.3.6 GetGlobalObject ()

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

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

8.4 Jobs and Job Queues

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

Execution of a Job can be initiated only when there is no running execution context and the execution context stack is empty. A PendingJob is a request for the future execution of a Job. A PendingJob is an internal Record whose fields are specified in Table 24. Once execution of a Job is initiated, the Job always executes to completion. No other Job may be initiated until the currently running Job completes. However, the currently running Job or external events may cause the enqueuing of additional PendingJobs that may be initiated sometime after completion of the currently running Job.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>

Table 24: PendingJob Record Fields
A Job Queue is a FIFO queue of PendingJob records. Each Job Queue has a name and the full set of available Job Queues are defined by an ECMAScript implementation. Every ECMAScript implementation has at least the Job Queues defined in Table 25.

Each agent has its own set of named Job Queues. All references to a named job queue in this specification denote the named job queue of the surrounding agent.

Table 25: Required Job Queues

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

A request for the future execution of a Job is made by enqueueing, on a Job Queue, a PendingJob record that includes a Job abstract operation name and any necessary argument values. When there is no running execution context and the execution context stack is empty, the ECMAScript implementation removes the first PendingJob from a Job Queue and uses the information contained in it to create an execution context and starts execution of the associated Job abstract operation.

The PendingJob records from a single Job Queue are always initiated in FIFO order. This specification does not define the order in which multiple Job Queues are serviced. An ECMAScript implementation may interweave the FIFO evaluation of the PendingJob records of a Job Queue with the evaluation of the PendingJob records of one or more other Job Queues. An implementation must define what occurs when there are no running execution context and all Job Queues are empty.
Typically an ECMAScript implementation will have its Job Queues pre-initialized with at least one PendingJob and one of those Jobs will be the first to be executed. An implementation might choose to free all resources and terminate if the current Job completes and all Job Queues are empty. Alternatively, it might choose to wait for a some implementation specific agent or mechanism to enqueue new PendingJob requests.

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

### 8.4.1 EnqueueJob (queueName, job, arguments)

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

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

### 8.5 InitializeHostDefinedRealm()

The abstract operation InitializeHostDefinedRealm performs the following steps:

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 an implementation-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 an implementation-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 implementation-defined global object properties on globalObj.

12. Return NormalCompletion(empty).

8.6 RunJobs ( )

The abstract operation RunJobs performs the following steps:

1. Perform ? InitializeHostDefinedRealm().
2. In an implementation-dependent manner, obtain the ECMAScript source texts (see clause 10) and any associated host-defined values for zero or more ECMAScript scripts and/or ECMAScript modules. For each such sourceText and hostDefined, do
   a. If sourceText is the source code of a script, then
   b. Else sourceText is the source code of a module,
      i. Perform EnqueueJob("Script Jobs", TopLevelModuleEvaluationJob, « sourceText, hostDefined »).
3. Repeat,
   a. SUSPEND the running execution context and remove it from the execution context stack.
   b. Assert: The execution context stack is now empty.
   c. Let nextQueue be a non-empty Job Queue chosen in an implementation-defined manner. If all Job Queues are empty, the result is implementation-defined.
   d. Let nextPending be the PendingJob record at the front of nextQueue. Remove that record from nextQueue.
   e. Let newContext be a new execution context.
   f. Set newContext's Function to null.
   g. Set newContext's Realm to nextPending.[[Realm]].
   h. Set newContext's ScriptOrModule to nextPending.[[ScriptOrModule]].
      i. Push newContext onto the execution context stack; newContext is now the running execution context.
   j. Perform any implementation or host environment defined job initialization using nextPending.
   k. Let result be the result of performing the abstract operation named by nextPending.[[Job]] using the elements of nextPending.[[Arguments]] as its arguments.
   l. If result is an abrupt completion, perform HostReportErrors(< result.[[Value ]] »).
8.7 Agents

An agent comprises a set of ECMAScript execution contexts, an execution context stack, a running execution context, a set of named job queues, 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 the jobs in the agent’s job queues 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 the jobs in the agent’s job queues, 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, the named job queues, and the Agent Record’s fields.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[LittleEndian]]</td>
<td>Boolean</td>
<td>The default value computed for the isLittleEndian parameter when it is needed by the algorithms GetValueFromBuffer and SetValueInBuffer. The choice is implementation-dependent 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>[[CanBlock]]</td>
<td>Boolean</td>
<td>Determines whether the agent can block or not.</td>
</tr>
<tr>
<td>[[Signifier]]</td>
<td>Any globally-unique value</td>
<td>Uniquely identifies the agent within its agent cluster.</td>
</tr>
<tr>
<td>[[IsLockFree1]]</td>
<td>Boolean</td>
<td>true if atomic operations on one-byte values are lock-free, false otherwise.</td>
</tr>
<tr>
<td>[[IsLockFree2]]</td>
<td>Boolean</td>
<td>true if atomic operations on two-byte values are lock-free, false otherwise.</td>
</tr>
<tr>
<td>[[CandidateExecution]]</td>
<td>A candidate</td>
<td>See the memory model.</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.

### 8.7.1 AgentSignifier()

The abstract operation AgentSignifier takes no arguments. It performs the following steps:

1. Let $AR$ be the Agent Record of the surrounding agent.
2. Return $AR[[\text{Signifier}]]$.

### 8.7.2 AgentCanSuspend()

The abstract operation AgentCanSuspend takes no arguments. It performs the following steps:

1. Let $AR$ be the Agent Record of the surrounding agent.
2. Return $AR[[\text{CanBlock}]]$.

**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.

## 8.8 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 [[EventLists]] field is a List containing, for
each agent, an Agent Events Record whose [[AgentSignifier]] field is that agent’s signifier and whose
[[EventList]] field is an empty List.

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.

8.9 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 Ordinary and Exotic Objects Behaviours

9.1 Ordinary Object Internal Methods and Internal Slots

All ordinary objects have an internal slot called [[Prototype]]. The value of this internal slot is either null
or an object and is used for implementing inheritance. Data properties of the [[Prototype]] object are
inherited (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 that controls whether or not
properties may be added to the object. If the value of the [[Extensible]] internal slot is false then additional
properties may not be added to the object. In addition, if [[Extensible]] is false the value of the
[[Prototype]] internal slot of the object may not be modified. Once the value of an object's [[Extensible]]
internal slot has been set to false it may not be subsequently changed to true.

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

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.

9.1.1 [[GetPrototypeOf]]

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

1. Return ! OrdinaryGetPrototypeOf($O$).

9.1.1.1 OrdinaryGetPrototypeOf($O$)

When the abstract operation OrdinaryGetPrototypeOf is called with Object $O$, the following steps are taken:
1. Return $O.\{\text{Prototype}\}$.

### 9.1.2 $\{\text{SetPrototypeOf}\} (V)$

When the $\{\text{SetPrototypeOf}\}$ internal method of $O$ is called with argument $V$, the following steps are taken:

1. Return $\text{OrdinarySetPrototypeOf}(O, V)$.

#### 9.1.2.1 OrdinarySetPrototypeOf ($O, V$)

When the abstract operation OrdinarySetPrototypeOf is called with Object $O$ and value $V$, the following steps are taken:

1. Assert: Either $\text{Type}(V)$ is Object or $\text{Type}(V)$ is Null.
2. Let $\text{extensible}$ be $O.\{\text{Extensible}\}$.
3. Let $\text{current}$ be $O.\{\text{Prototype}\}$.
4. If $\text{SameValue}(V, \text{current})$ is $\text{true}$, return $\text{true}$.
5. If $\text{extensible}$ is $\text{false}$, return $\text{false}$.
7. Let $\text{done}$ be $\text{false}$.
8. Repeat, while $\text{done}$ is $\text{false}$,
   a. If $p$ is $\text{null}$, set $\text{done}$ to $\text{true}$.
   b. Else if $\text{SameValue}(p, O)$ is $\text{true}$, return $\text{false}$.
   c. Else,
      i. If $p.\{\text{GetPrototypeOf}\}$ is not the ordinary object internal method defined in 9.1.1, set $\text{done}$ to $\text{true}$.
      ii. Else, set $p$ to $p.\{\text{Prototype}\}$.
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 $\{\text{SetPrototypeOf}\}$.

### 9.1.3 $\{\text{IsExtensible}\} ()$

When the $\{\text{IsExtensible}\}$ internal method of $O$ is called, the following steps are taken:

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

#### 9.1.3.1 OrdinaryIsExtensible ($O$)
When the abstract operation OrdinaryIsExtensible is called with Object \( O \), the following steps are taken:

1. Return \( O.\{\text{Extensible}\} \).

### 9.1.4 \texttt{[[PreventExtensions]]} ( )

When the \texttt{[[PreventExtensions]]} internal method of \( O \) is called, the following steps are taken:

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

#### 9.1.4.1 OrdinaryPreventExtensions ( \( O \) )

When the abstract operation OrdinaryPreventExtensions is called with Object \( O \), the following steps are taken:

1. Set \( O.\{\text{Extensible}\} \) to \texttt{false}.
2. Return \texttt{true}.

### 9.1.5 \texttt{[[GetOwnProperty]]} ( \( P \) )

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

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

#### 9.1.5.1 OrdinaryGetOwnProperty ( \( O, P \) )

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

1. Assert: \texttt{IsPropertyKey(P)} is \texttt{true}.
2. If \( O \) does not have an own property with key \( P \), return \texttt{undefined}.
3. Let \( D \) be a newly created \texttt{PropertyDescriptor} 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.
4. Else \( X \) is an accessor property,
   a. Set \( D.[\{\text{Get}\}] \) to the value of \( X \)'s \{Get\} attribute.
   b. 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.

### 9.1.6 [[DefineOwnProperty]] ($P$, $Desc$)

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


### 9.1.6.1 OrdinaryDefineOwnProperty ($O$, $P$, $Desc$)

When the abstract operation ` OrdinaryDefineOwnProperty` is called with Object $O$, property key $P$, and Property Descriptor $Desc$, the following steps are taken:

2. Let `extensible` be $O$. [[Extensible]].
3. Return `ValidateAndApplyPropertyDescriptor($O$, $P$, `extensible`, $Desc$, `current`).`

### 9.1.6.2 IsCompatiblePropertyDescriptor ($Extensible$, $Desc$, $Current$)

When the abstract operation `IsCompatiblePropertyDescriptor` is called with Boolean value $Extensible$, and Property Descriptors $Desc$, and $Current$, the following steps are taken:

1. Return `ValidateAndApplyPropertyDescriptor(undefined, undefined, Extensible, Desc, Current).`.

### 9.1.6.3 ValidateAndApplyPropertyDescriptor ($O$, $P$, `extensible`, $Desc$, $current$)

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

**NOTE**

If `undefined` is passed as the $O$ argument only validation is performed and no object updates are performed.

1. Assert: If $O$ is not `undefined`, then `IsPropertyKey($P$)` is `true`.
2. If `current` is `undefined`, then
   a. If `extensible` is `false`, return `false`.
   b. Assert: `extensible` is `true`.
   c. If `IsGenericDescriptor($Desc$)` is `true` or `IsDataDescriptor($Desc$)` is `true`, then
      i. If $O$ is not `undefined`, create an own data property named $P$ of object $O$ whose `[[Value]]`, `[[Writable]]`, `[[Enumerable]]` and `[[Configurable]]` attribute values are described by $Desc$. If the value of an attribute field of $Desc$ is absent, the attribute of the newly created property is set to its default value.
d. Else Desc must be an accessor Property Descriptor,
   i. 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 the [[Enumerable]] fields of current and Desc are the
      Boolean negation of each other, return false.

5. If IsGenericDescriptor(Desc) is true, no further validation is required.

6. Else if IsDataDescriptor(current) and IsDataDescriptor(Desc) have different results, 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 IsAccessorDescriptor(current) and IsAccessorDescriptor(Desc) are both true, then
   a. 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.
9.1.7 [[HasProperty]](P)

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


9.1.7.1 OrdinaryHasProperty (O, P)

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

1. Assert: IsPropertyKey(P) is true.
2. Let hasOwn be O.[[GetOwnProperty]](P).
3. If hasOwn is not undefined, return true.
4. Let parent be O.[[GetPrototypeOf]]().
5. If parent is not null, then
6. Return false.

9.1.8 [[Get]] (P, Receiver)

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


9.1.8.1 OrdinaryGet (O, P, Receiver)

When the abstract operation OrdinaryGet is called with Object O, property key P, and ECMAScript language value Receiver, the following steps are taken:

1. Assert: IsPropertyKey(P) is true.
2. Let desc be O.[[GetOwnProperty]](P).
3. If desc is undefined, then
   a. Let parent be O.[[GetPrototypeOf]]().
   b. If parent is null, return undefined.
4. If IsDataDescriptor(desc) is true, return desc.[[Value]].
5. Assert: IsAccessorDescriptor(desc) is true.
6. Let getter be desc.[[Get]].
7. If getter is undefined, return undefined.
9.1.9  [[Set]] (P, V, Receiver)

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


9.1.9.1 OrdinarySet (O, P, V, Receiver)

When the abstract operation OrdinarySet is called with Object O, property key P, value V, and ECMAScript language value Receiver, the following steps are taken:

1. Assert: IsPropertyKey(P) is true.
3. If ownDesc is undefined, then
   a. Let parent be ? O.[[GetPrototypeOf]]().
   b. If parent is not null, then
   c. Else,
      i. Set ownDesc to the PropertyDescriptor{[[Value]]: undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
4. If IsDataDescriptor(ownDesc) is true, then
   a. If ownDesc.[[Writable]] is false, return false.
   b. If Type(Receiver) is not Object, return false.
   c. Let existingDescriptor be ? Receiver.[[GetOwnProperty]](P).
   d. If existingDescriptor is not undefined, then
      i. If IsAccessorDescriptor(existingDescriptor) is true, return false.
      ii. If existingDescriptor.[[Writable]] is false, return false.
      iii. Let valueDesc be the PropertyDescriptor{[[Value]]: V}.
   e. Else Receiver does not currently have a property P,
      i. Return ? CreateDataProperty(Receiver, P, V).
5. Assert: IsAccessorDescriptor(ownDesc) is true.
6. Let setter be ownDesc.[[Set]].
7. If setter is undefined, return false.
8. Perform ? Call(setter, Receiver, « V »).
9. Return true.

9.1.10  [[Delete]] (P)

When the [[Delete]] internal method of O is called with property key P, the following steps are taken:
1. Return \(? OrdinaryDelete(O, P).\\)

### 9.1.10.1 OrdinaryDelete \((O, P)\\)

When the abstract operation OrdinaryDelete is called with Object \(O\) and property key \(P\), the following steps are taken:

1. Assert: \(\text{IsPropertyKey}(P)\) is \(true\).
2. Let \(desc\) be \(? O.\text{[GetOwnProperty]}(P)\).
3. If \(desc\) is \(\text{undefined}\), return \(true\).
4. If \(desc.\text{[[Configurable]]}\) is \(true\), then
   a. Remove the own property with name \(P\) from \(O\).
   b. Return \(true\).
5. Return \(false\).

### 9.1.11 \([\text{OwnPropertyKeys}]()\)\\

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

1. Return \(! OrdinaryOwnPropertyKeys(O)\).

### 9.1.11.1 OrdinaryOwnPropertyKeys \((O)\\)

When the abstract operation OrdinaryOwnPropertyKeys is called with Object \(O\), the following steps are taken:

1. Let \(keys\) be a new empty \(\text{List}\).
2. For each own property key \(P\) of \(O\) that is an integer index, in ascending numeric index order, do
   a. Add \(P\) as the last element of \(keys\).
3. For each own property key \(P\) of \(O\) that is a String but is not an integer index, in ascending chronological order of property creation, do
   a. Add \(P\) as the last element of \(keys\).
4. For each own property key \(P\) of \(O\) that is a Symbol, in ascending chronological order of property creation, do
   a. Add \(P\) as the last element of \(keys\).
5. Return \(keys\).

### 9.1.12 ObjectCreate \((proto [, internalSlotsList ]))\\

The abstract operation ObjectCreate with argument \(proto\) (an object or null) is used to specify the runtime creation of new ordinary objects. The optional argument \(internalSlotsList\) is a \(\text{List}\) of the names of
additional internal slots that must be defined as part of the object. If the list is not provided, a new empty `List` is used. This abstract operation performs the following steps:

1. If `internalSlotsList` was not provided, set `internalSlotsList` to a new empty `List`.
2. Let `obj` be a newly created object with an internal slot for each name in `internalSlotsList`.
3. Set `obj`'s essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set `obj`'s `[[Prototype]]` to `proto`.
5. Set `obj`'s `[[Extensible]]` to `true`.
6. Return `obj`.

### 9.1.13 OrdinaryCreateFromConstructor (`constructor, intrinsicDefaultProto [, internalSlotsList]`)

The abstract operation `OrdinaryCreateFromConstructor` creates an ordinary object whose `[[Prototype]]` value is retrieved from a constructor's `prototype` property, if it exists. Otherwise the intrinsic named by `intrinsicDefaultProto` is used for `[[Prototype]]`. The optional `internalSlotsList` is a `List` of the names of additional internal slots that must be defined as part of the object. If the list is not provided, a new empty `List` is used. This abstract operation performs the following steps:

1. Assert: `intrinsicDefaultProto` is a String value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the `[[Prototype]]` value of an object.
2. Let `proto` be ? `GetPrototypeFromConstructor(constructor, intrinsicDefaultProto)`.
3. Return `ObjectCreate(proto, internalSlotsList)`.

### 9.1.14 GetPrototypeFromConstructor (`constructor, intrinsicDefaultProto`)

The abstract operation `GetPrototypeFromConstructor` 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]]`. This abstract operation performs the following steps:

1. Assert: `intrinsicDefaultProto` is a String value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the `[[Prototype]]` value of an object.
2. 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.

### 9.2 ECMAScript Function Objects

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

ECMAScript function objects have the additional internal slots listed in Table 27.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Environment]]</code></td>
<td>Lexical Environment</td>
<td>The Lexical Environment that the function was closed over. Used as the outer environment when evaluating the code of the function.</td>
</tr>
<tr>
<td><code>[[FormalParameters]]</code></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><code>[[FunctionKind]]</code></td>
<td>String</td>
<td>Either &quot;normal&quot;, &quot;classConstructor&quot;, &quot;generator&quot;, or &quot;async&quot;.</td>
</tr>
<tr>
<td><code>[[ECMAScriptCode]]</code></td>
<td>Parse Node</td>
<td>The root parse node of the source text that defines the function's body.</td>
</tr>
<tr>
<td><code>[[ConstructorKind]]</code></td>
<td>String</td>
<td>Either &quot;base&quot; or &quot;derived&quot;.</td>
</tr>
<tr>
<td><code>[[Realm]]</code></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><code>[[ScriptOrModule]]</code></td>
<td>Script Record or Module Record</td>
<td>The script or module in which the function was created.</td>
</tr>
<tr>
<td><code>[[ThisMode]]</code></td>
<td>(lexical, strict,</td>
<td>Defines how <code>this</code> references are interpreted within the formal parameters and code body of the function. <code>lexical</code> means that</td>
</tr>
</tbody>
</table>

---
**Global**

**Strict** means that the **this** value is used exactly as provided by an invocation of the function. **Global** means that a **this** value of **undefined** is interpreted as a reference to the **global** object.

| **Strict** | **Boolean** | **true** if this is a **strict** function, **false** if this is a **non-strict** function. |
| **HomeObject** | **Object** | If the function uses **super**, this is the object whose **[GetPrototypeOf]** provides the object where **super** property lookups begin. |

All ECMAScript function objects have the **[[Call]]** internal method defined here. ECMAScript functions that are also constructors in addition have the **[[Construct]]** internal method.

### 9.2.1 **[[Call]] (thisArgument, argumentsList)**

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

1. **Assert**: **F** is an ECMAScript function object.
2. **If** **F.**[[FunctionKind]] is "**classConstructor**", throw a **TypeError** exception.
3. **Let** **calleeContext** be the running execution context.
4. **Let** **calleeContext** be **PrepareForOrdinaryCall**( **F**, **undefined**).
5. **Assert**: **calleeContext** is now the running execution context.
6. **Perform** **OrdinaryCallBindThis**( **F**, **calleeContext**, **thisArgument**).
7. **Let** **result** be **OrdinaryCallEvaluateBody**( **F**, **argumentsList**).
8. **Remove** **calleeContext** from the execution context stack and restore **calleeContext** 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.

### 9.2.1.1 **PrepareForOrdinaryCall (F, newTarget)**

When the abstract operation **PrepareForOrdinaryCall** is called with function object **F** and **ECMAScript** language value **newTarget**, the following steps are taken:
1. Assert: Type(newTarget) is Undefined or Object.
2. Let callerContext be the running execution context.
3. Let calleeContext be a new ECMAScript code execution context.
4. Set the Function of calleeContext to F.
5. Let calleeRealm be F.[[Realm]].
6. Set the Realm of calleeContext to calleeRealm.
7. Set the ScriptOrModule of calleeContext to F.[[ScriptOrModule]].
8. Let localEnv be NewFunctionEnvironment(F, newTarget).
9. Set the LexicalEnvironment of calleeContext to localEnv.
10. Set the VariableEnvironment of calleeContext to localEnv.
11. If callerContext is not already suspended, suspend callerContext.
12. Push calleeContext onto the execution context stack; calleeContext is now the running execution context.
13. NOTE: Any exception objects produced after this point are associated with calleeRealm.

9.2.1.2 OrdinaryCallBindThis (F, calleeContext, thisArgument)

When the abstract operation OrdinaryCallBindThis is called with function object F, execution context calleeContext, and ECMAScript value thisArgument, the following steps are taken:

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. Let globalEnvRec be globalEnv's EnvironmentRecord.
      iii. Assert: globalEnvRec is a global Environment Record.
      iv. Let thisValue be globalEnvRec.[[GlobalThisValue]].
   b. Else,
      i. Let thisValue be ! ToObject(thisArgument).
      ii. NOTE: ToObject produces wrapper objects using calleeRealm.
7. Let envRec be localEnv's EnvironmentRecord.
8. Assert: envRec is a function Environment Record.
9. Assert: The next step never returns an abrupt completion because envRec.[[ThisBindingStatus]] is not "initialized".
10. Return envRec.BindThisValue(thisValue).
9.2.1.3 **OrdinaryCallEvaluateBody (F, argumentsList)**

When the abstract operation `OrdinaryCallEvaluateBody` is called with function object \( F \) and List \( \text{argumentsList} \), the following steps are taken:

1. Return the result of `EvaluateBody` of the parsed code that is \( F.\llbracket \text{ECMAScriptCode} \rrbracket \) passing \( F \) and \( \text{argumentsList} \) as the arguments.

9.2.2 **[[Construct]] (argumentsList, newTarget)**

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

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

9.2.3 **FunctionAllocate (functionPrototype, strict, functionKind)**

The abstract operation `FunctionAllocate` requires the three arguments \( \text{functionPrototype} \), \( \text{strict} \) and \( \text{functionKind} \). FunctionAllocate performs the following steps:

1. Assert: \( \text{Type(functionPrototype)} \) is Object.
2. Assert: \( \text{functionKind} \) is either "normal", "non-constructor", "generator", or "async".
3. If `functionKind` is "normal", let `needsConstruct` be `true`.
4. Else, let `needsConstruct` be `false`.
5. If `functionKind` is "non-constructor", set `functionKind` to "normal".
6. Let $F$ be a newly created ECMAScript function object with the internal slots listed in Table 27. All of those internal slots are initialized to `undefined`.
7. Set $F$'s essential internal methods to the default ordinary object definitions specified in 9.1.
8. Set $F$.[[Call]] to the definition specified in 9.2.1.
9. If `needsConstruct` is `true`, then
   a. Set $F$.[[Construct]] to the definition specified in 9.2.2.
   b. Set $F$.[[ConstructorKind]] to "base".
10. Set $F$.[[Strict]] to `strict`.
11. Set $F$.[[FunctionKind]] to `functionKind`.
12. Set $F$.[[Prototype]] to `functionPrototype`.
13. Set $F$.[[Extensible]] to `true`.
14. Set $F$.[[Realm]] to the current Realm Record.
15. Return $F$.

### 9.2.4 FunctionInitialize ( $F$, `kind`, `ParameterList`, `Body`, `Scope` )

The abstract operation FunctionInitialize requires the arguments: a function object $F$, `kind` which is one of (Normal, Method, Arrow), a parameter list Parse Node specified by `ParameterList`, a body Parse Node specified by `Body`, a Lexical Environment specified by `Scope`. FunctionInitialize performs the following steps:

1. Assert: $F$ is an extensible object that does not have a `length` own property.
2. Let `len` be the ExpectedArgumentCount of `ParameterList`.
3. Perform ! DefinePropertyOrThrow($F$, "length", PropertyDescriptor{[[Value]]: `len`, [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `true`}).
4. Let `Strict` be $F$.[[Strict]].
5. Set $F$.[[Environment]] to `Scope`.
6. Set $F$.[[FormalParameters]] to `ParameterList`.
7. Set $F$.[[ECMAScriptCode]] to `Body`.
8. Set $F$.[[ScriptOrModule]] to GetActiveScriptOrModule().
9. If `kind` is Arrow, set $F$.[[ThisMode]] to lexical.
10. Else if `Strict` is `true`, set $F$.[[ThisMode]] to strict.
11. Else, set $F$.[[ThisMode]] to global.
12. Return $F$.

### 9.2.5 FunctionCreate ( `kind`, `ParameterList`, `Body`, `Scope`, `Strict`, `prototype` )
The abstract operation FunctionCreate requires the arguments: kind which is one of (Normal, Method, Arrow), a parameter list Parse Node specified by ParameterList, a body Parse Node specified by Body, a Lexical Environment specified by Scope, a Boolean flag Strict, and optionally, an object prototype. FunctionCreate performs the following steps:

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

9.2.6 GeneratorFunctionCreate (kind, ParameterList, Body, Scope, Strict)

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

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

9.2.7 AddRestrictedFunctionProperties (F, realm)

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

1. Assert: realm.([Intrinsics]).[[%ThrowTypeError%]] exists and has been initialized.
2. Let thrower be realm.([Intrinsics]).[[%ThrowTypeError%]].
3. Perform ! DefinePropertyOrThrow(F, "caller", PropertyDescriptor {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: true}).
4. Return ! DefinePropertyOrThrow(F, "arguments", PropertyDescriptor {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: true}).

9.2.7.1 %ThrowTypeError%( )

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

1. Throw a TypeError exception.
The value of the [[Extensible]] internal slot of a %ThrowTypeError% function is false.

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

9.2.8 MakeConstructor ( F [, writablePrototype [, prototype ] ] )

The abstract operation MakeConstructor requires a Function argument F and optionally, a Boolean writablePrototype and an object prototype. If prototype is provided it is assumed to already contain, if needed, a "constructor" property whose value is F. This operation converts F into a constructor by performing the following steps:

1. Assert: F is an ECMAScript function object.
2. Assert: F has a [[Construct]] internal method.
3. Assert: F is an extensible object that does not have a prototype own property.
4. If the writablePrototype argument was not provided, set writablePrototype to true.
5. If the prototype argument was not provided, then
   a. Set prototype to ObjectCreate(%ObjectPrototype%).
   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).

9.2.9 MakeClassConstructor ( F )

The abstract operation MakeClassConstructor with argument F performs the following steps:

1. Assert: F is an ECMAScript function object.
2. Assert: F.[[FunctionKind]] is "normal".
3. Set F.[[FunctionKind]] to "classConstructor".
4. Return NormalCompletion(undefined).

9.2.10 MakeMethod ( F, homeObject )

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

1. Assert: F is an ECMAScript function object.
2. Assert: Type(homeObject) is Object.
3. Set F.[[HomeObject]] to homeObject.
4. Return `NormalCompletion(undefined)`.

### 9.2.11 SetFunctionName (\(F, name [ , prefix ]\))

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

1. Assert: \(F\) is an extensible object that does not have a \(name\) own property.
2. Assert: \(Type(name)\) is either Symbol or String.
3. Assert: If \(prefix\) was passed, then \(Type(prefix)\) is String.
4. If \(Type(name)\) is Symbol, then
   a. Let \(description\) be \(name\)'s \([[\text{Description}]]\) value.
   b. If \(description\) is \(undefined\), set \(name\) to the empty String.
   c. Else, set \(name\) to the concatenation of "[", \(description\), and "]".
5. If \(prefix\) was passed, then
   a. Set \(name\) to the concatenation of \(prefix\), code unit 0x0020 (SPACE), and \(name\).
6. Return ! `DefinePropertyOrThrow(F, "name", PropertyDescriptor{[[Value]]: name, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}).`

### 9.2.12 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.

FunctionDeclarationInstantiation is performed as follows using arguments \(func\) and \(argumentsList\). \(func\) is the function object for which the execution context is being established.

1. Let \(calleeContext\) be the running execution context.
2. Let \(env\) be the LexicalEnvironment of \(calleeContext\).
3. Let \(envRec\) be \(env\)'s EnvironmentRecord.
4. Let \(code\) be \(func.[[\text{ECMAScriptCode}]]\).
5. Let \(strict\) be \(func.[[\text{Strict}]]\).
6. Let \(formals\) be \(func.[[\text{FormalParameters}]]\).
7. Let `parameterNames` be the BoundNames of `formals`.
8. If `parameterNames` has any duplicate entries, let `hasDuplicates` be true. Otherwise, let `hasDuplicates` be false.
10. Let `hasParameterExpressions` be ContainsExpression of `formals`.
11. Let `varNames` be the VarDeclaredNames of `code`.
12. Let `varDeclarations` be the VarScopedDeclarations of `code`.
13. Let `lexicalNames` be the LexicallyDeclaredNames of `code`.
14. Let `functionNames` be a new empty List.
15. Let `functionsToInitialize` be a new empty List.
16. For each `d` in `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`, or an `AsyncFunctionDeclaration`.
       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`.
17. Let `argumentsObjectNeeded` be true.
18. If `func.[[ThisMode]]` is lexical, then
    a. NOTE: Arrow functions never have an arguments objects.
    b. Set `argumentsObjectNeeded` to false.
19. Else if "arguments" is an element of `parameterNames`, then
    a. Set `argumentsObjectNeeded` to false.
20. 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.
21. For each String `paramName` in `parameterNames`, do
    a. Let `alreadyDeclared` be `envRec.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 ! `envRec.CreateMutableBinding(paramName, false)`.
       ii. If `hasDuplicates` is true, then
           1. Perform ! `envRec.InitializeBinding(paramName, undefined)`.
22. If `argumentsObjectNeeded` is true, then
    a. If `strict` is true or if `simpleParameterList` is false, then
i. Let \( ao \) be \( \text{CreateUnmappedArgumentsObject}(argumentsList) \).

b. Else,
   i. NOTE: mapped argument object is only provided for non-strict functions that don't have a rest parameter, any parameter default value initializers, or any destructured parameters.
   ii. Let \( ao \) be \( \text{CreateMappedArgumentsObject}(func, \text{formals}, argumentsList, \text{envRec}) \).

c. If \( \text{strict} \) is \( \text{true} \), then
   i. Perform \( ! \text{envRec}.\text{CreateImmutableBinding}(\"arguments\", \text{false}) \).

d. Else,
   i. Perform \( ! \text{envRec}.\text{CreateMutableBinding}(\"arguments\", \text{false}) \).
   e. Call \( \text{envRec}.\text{InitializeBinding}(\"arguments\", \text{ao}) \).
   f. Append \( \"arguments\" \) to \( \text{parameterNames} \).

23. Let \( \text{iteratorRecord} \) be \( \text{Record} \{[[\text{Iterator}]]: \text{CreateListIterator}(argumentsList), [[\text{Done}]]: \text{false}\} \).

24. If \( \text{hasDuplicates} \) is \( \text{true} \), then
   a. Perform \( ? \text{IteratorBindingInitialization} \) for \( \text{formals} \) with \( \text{iteratorRecord} \) and \( \text{undefined} \) as arguments.

25. Else,
   a. Perform \( ? \text{IteratorBindingInitialization} \) for \( \text{formals} \) with \( \text{iteratorRecord} \) and \( \text{env} \) as arguments.

26. If \( \text{hasParameterExpressions} \) is \( \text{false} \), then
   a. NOTE: Only a single lexical environment is needed for the parameters and top-level vars.
   b. Let \( \text{instantiatedVarNames} \) be a copy of the \( \text{List parameterNames} \).
   c. For each \( n \) in \( \text{varNames} \), do
      i. If \( n \) is not an element of \( \text{instantiatedVarNames} \), then
         1. Append \( n \) to \( \text{instantiatedVarNames} \).
         2. Perform \( ! \text{envRec}.\text{CreateMutableBinding}(n, \text{false}) \).
         3. Call \( \text{envRec}.\text{InitializeBinding}(n, \text{undefined}) \).
   d. Let \( \text{varEnv} \) be \( \text{env} \).
   e. Let \( \text{varEnvRec} \) be \( \text{envRec} \).

27. 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 \( \text{varEnv} \) be \( \text{NewDeclarativeEnvironment(env)} \).
   c. Let \( \text{varEnvRec} \) be \( \text{varEnv} \)'s EnvironmentRecord.
   d. Set the VariableEnvironment of \( \text{calleeContext} \) to \( \text{varEnv} \).
   e. Let \( \text{instantiatedVarNames} \) be a new empty \( \text{List} \).
   f. For each \( n \) in \( \text{varNames} \), do
      i. If \( n \) is not an element of \( \text{instantiatedVarNames} \), then
         1. Append \( n \) to \( \text{instantiatedVarNames} \).
         2. Perform \( ! \text{varEnvRec}.\text{CreateMutableBinding}(n, \text{false}) \).
3. If \( n \) is not an element of \( \text{parameterNames} \) or if \( n \) is an element of \( \text{functionNames} \), let \( \text{initialValue} \) be \text{undefined}.

4. Else,
   a. Let \( \text{initialValue} \) be \( \text{! envRec.GetBindingValue} (n, \text{false}) \).

5. Call \( \text{varEnvRec.InitializeBinding} (n, \text{initialValue}) \).

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

28. NOTE: Annex B.3.3.1 adds additional steps at this point.

29. If \( \text{strict} \) is \text{false}, then
   a. Let \( \text{lexEnv} \) be \( \text{NewDeclarativeEnvironment} (\text{varEnv}) \).
   b. NOTE: Non-strict functions use a separate lexical Environment Record for top-level lexical declarations so that a \text{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 \text{direct eval} always places all declarations into a new Environment Record.

30. Else, let \( \text{lexEnv} \) be \( \text{varEnv} \).

31. Let \( \text{lexEnvRec} \) be \( \text{lexEnv}'s \) EnvironmentRecord.

32. Set the LexicalEnvironment of \( \text{calleeContext} \) to \( \text{lexEnv} \).

33. Let \( \text{lexDeclarations} \) be the Lexically Scoped Declarations of \( \text{code} \).

34. For each element \( d \) in \( \text{lexDeclarations} \), 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 \text{true}, then
         1. Perform \( \text{! lexEnvRec.CreateImmutableBinding} (dn, \text{true}) \).
      ii. Else,
         1. Perform \( \text{! lexEnvRec.CreateMutableBinding} (dn, \text{false}) \).

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

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 \text{direct eval} expressions. Any top level declarations of such evals are only visible to the eval code (10.2). The creation of the environment for such declarations is described in 14.1.19.
9.3 Built-in Function Objects

The built-in function objects defined in this specification may be implemented as either ECMAScript function objects (9.2) whose behaviour is provided using ECMAScript code or as implementation provided 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 9.1. All such function exotic objects also have [[Prototype]], [[Extensible]], [[Realm]], and [[ScriptOrModule]] internal slots.

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

The behaviour specified for each built-in function via algorithm steps or other means is the specification of the function body behaviour for both [[Call]] and [[Construct]] invocations of the function. However, [[Construct]] invocation is not supported by all built-in functions. For each built-in function, when invoked with [[Call]], the [[Call]] thisArgument provides the this value, the [[Call]] argumentsList provides the named parameters, and the NewTarget value is undefined. When invoked with [[Construct]], the this value is uninitialized, the [[Construct]] argumentsList provides the named parameters, and the [[Construct]] newTarget parameter provides the NewTarget value. If the built-in function is implemented as an ECMAScript function object then this specified behaviour must be implemented by the ECMAScript code that is the body of the function. Built-in functions that are ECMAScript function objects must be strict functions. If a built-in constructor has any [[Call]] behaviour other than throwing a TypeError exception, an ECMAScript implementation of the function must be done in a manner that does not cause the function’s [[FunctionKind]] internal slot to have the value "classConstructor".

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

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

If a built-in function object is not implemented as an ECMAScript function it must provide [[Call]] and [[Construct]] internal methods that conform to the following definitions:

9.3.1 [[Call]] (thisArgument, argumentsList)
The [[Call]] internal method for a built-in function object $F$ is called with parameters `thisArgument` and `argumentsList`, a List of ECMAScript language values. The following steps are taken:

1. Let `calleeContext` be the running execution context.
2. If `calleeContext` is not already suspended, suspend `calleeContext`.
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. 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 an implementation-defined 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 `calleeContext` 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.

### 9.3.2 [[Construct]] (argumentsList, newTarget)

The [[Construct]] internal method for built-in function object $F$ is called with parameters `argumentsList` and `newTarget`. The steps performed are the same as [[Call]] (see 9.3.1) except that step 10 is replaced by:

10. Let `result` be the Completion Record that is the result of evaluating $F$ in an implementation-defined manner that conforms to the specification of $F$. The `this` value is uninitialized, `argumentsList` provides the named parameters, and `newTarget` provides the NewTarget value.

### 9.3.3 CreateBuiltInFunction (realm, steps, prototype [, internalSlotsList ])

The abstract operation CreateBuiltInFunction takes arguments `realm, steps, prototype`. The optional argument `internalSlotsList` is a List of the names of additional internal slots that must be defined as part of the object. If the list is not provided, a new empty List is used. CreateBuiltInFunction returns a built-in function object created by the following steps:

1. Assert: `realm` is a Realm Record.
2. Assert: steps is either a set of algorithm steps or other definition of a function’s behaviour provided in this specification.

3. 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. The initial value of each of those internal slots is undefined.

4. Set func. [ [ Realm ]] to realm.
5. Set func. [ [ Prototype ]] to prototype.
7. Set func. [ [ ScriptOrModule ]] to null.
8. Return func.

Each built-in function defined in this specification is created as if by calling the CreateBuiltInFunction abstract operation, unless otherwise specified.

9.4 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:

9.4.1 Bound Function Exotic Objects

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

Bound function objects do not have the internal slots of ECMAScript function objects defined in Table 27. Instead they have the internal slots defined in Table 28.

<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>
Bound function objects provide all of the essential internal methods as specified in 9.1. However, they use the following definitions for the essential internal methods of function objects.

9.4.1.1  [[Call]] (thisArgument, argumentsList)

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

1. Let target be \( F.[[BoundTargetFunction]] \).
2. Let boundThis be \( F.[[BoundThis]] \).
3. Let boundArgs be \( F.[[BoundArguments]] \).
4. Let args be a new list containing the same values as the list boundArgs in the same order followed by the same values as the list argumentsList in the same order.
5. Return ? Call(target, boundThis, args).

9.4.1.2  [[Construct]] (argumentsList, newTarget)

When the [[Construct]] internal method of a bound function exotic object, \( F \) that was created using the bind function is called with a list of arguments argumentsList and newTarget, the following steps are taken:

1. Let target be \( F.[[BoundTargetFunction]] \).
2. Assert: target has a [[Construct]] internal method.
3. Let boundArgs be \( F.[[BoundArguments]] \).
4. Let args be a new list containing the same values as the list boundArgs in the same order followed by the same values as the list argumentsList in the same order.
5. If SameValue(F, newTarget) is true, set newTarget to target.

9.4.1.3  BoundFunctionCreate (targetFunction, boundThis, boundArgs)

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

1. Assert: Type(targetFunction) is Object.
2. Let proto be ? targetFunction.[[GetPrototypeOf]]().
3. Let obj be a newly created object.
4. Set obj's essential internal methods to the default ordinary object definitions specified in 9.1.
5. Set obj.[[Call]] as described in 9.4.1.1.
6. If targetFunction has a [[Construct]] internal method, then
   a. Set obj.[[Construct]] as described in 9.4.1.2.
7. Set obj.[[Prototype]] to proto.
8. Set `obj.[[Extensible]]` to `true`.
10. Set `obj.[[BoundThis]]` to `boundThis`.
11. Set `obj.[[BoundArguments]]` to `boundArgs`.
12. Return `obj`.

### 9.4.2 Array Exotic Objects

An *Array object* is an exotic object that gives special treatment to array index property keys (see 6.1.7). A property whose property name is an array index is also called an *element*. Every Array object has a `length` property whose value is always a nonnegative integer less than $2^{32}$. The value of the `length` property is numerically greater than the name of every 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))` is equal to `P` and `ToUint32(P)` is not equal to $2^{32}-1$.

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

Array exotic objects provide an alternative definition for the `[[DefineOwnProperty]]` internal method. Except for that internal method, Array exotic objects provide all of the other essential internal methods as specified in 9.1.

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

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

1. Assert: `IsPropertyKey(P)` is `true`.
2. If `P` is "length", then
3. Else if `P` is an array index, then
   a. Let `oldLenDesc` be `OrdinaryGetOwnProperty(A, "length")`.
   b. Assert: `oldLenDesc` will never be `undefined` or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.
c. Let oldLen be oldLenDesc.[[Value]].

d. Let index be ! ToUint32(P).

e. If index ≥ oldLen and oldLenDesc.[[Writable]] is false, return false.

f. Let succeeded be ! OrdinaryDefineOwnProperty(A, P, Desc).

g. If succeeded is false, return false.

h. If index ≥ oldLen, then
   i. Set oldLenDesc.[[Value]] to index + 1.
   ii. Let succeeded be OrdinaryDefineOwnProperty(A, "length", oldLenDesc).
      iii. Assert: succeeded is true.
   i. Return true.


### 9.4.2.2 ArrayCreate (length [, proto ])

The abstract operation ArrayCreate with argument length (either 0 or a positive integer) and optional argument proto is used to specify the creation of new Array exotic objects. It performs the following steps:

1. Assert: length is an integer Number ≥ 0.
2. If length is -0, set length to +0.
3. If length>2^{32}-1, throw a RangeError exception.
4. If the proto argument was not passed, set proto to the intrinsic object %ArrayPrototype%.
5. Let A be a newly created Array exotic object.
6. Set A's essential internal methods except for [[DefineOwnProperty]] to the default ordinary object definitions specified in 9.1.
7. Set A.[[DefineOwnProperty]] as specified in 9.4.2.1.
8. Set A.[[Prototype]] to proto.
9. Set A.[[Extensible]] to true.
10. Perform ! OrdinaryDefineOwnProperty(A, "length", PropertyDescriptor{[[Value]]: length,
                            [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}).
11. Return A.

### 9.4.2.3 ArraySpeciesCreate (originalArray, length)

The abstract operation ArraySpeciesCreate with arguments originalArray and length 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:

1. Assert: length is an integer Number ≥ 0.
2. If length is -0, set length to +0.
3. Let isArray be ? IsArray(originalArray).
4. If isArray is false, return ? ArrayCreate(length).
5. Let $C$ be ? Get($originalArray$, "constructor").
6. 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.
7. If Type($C$) is Object, then
   a. Set $C$ to ? Get($C$, @@species).
   b. If $C$ is null, set $C$ to undefined.
8. If $C$ is undefined, return ? ArrayCreate(length).
9. If IsConstructor($C$) is false, throw a TypeError exception.

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.

9.4.2.4 ArraySetLength ($A$, Desc)

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

1. If Desc.[[Value]] is absent, then
2. Let newLenDesc be a copy of Desc.
3. Let newLen be ? ToUint32(Desc.{{Value}}).
4. Let numberLen be ? ToNumber(Desc.{{Value}}).
5. If newLen ≠ numberLen, throw a RangeError exception.
7. Let oldLenDesc be OrdinaryGetOwnProperty($A$, "length").
8. Assert: oldLenDesc will never be undefined or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.
9. Let oldLen be oldLenDesc.{{Value}}.
10. If newLen ≥ oldLen, then
11. If oldLenDesc.{{Writable}} is false, return false.
12. If newLenDesc.{{Writable}} is absent or has the value true, let newWritable be true.
13. Else,
    a. Need to defer setting the {{Writable}} attribute to false in case any elements cannot be deleted.
b. Let newWritable be false.
c. Set newLenDesc.[[Writable]] to true.
15. If succeeded is false, return false.
16. Repeat, while newLen < oldLen,
   a. Set oldLen to oldLen - 1.
   b. Let deleteSucceeded be ! A.[[Delete]](! ToString(oldLen)).
   c. If deleteSucceeded is false, then
      i. Set newLenDesc.[[Value]] to oldLen + 1.
      ii. If newWritable is false, set newLenDesc.[[Writable]] to false.
      iii. Let succeeded be ! OrdinaryDefineOwnProperty(A, "length", newLenDesc).
      iv. Return false.
17. If newWritable is false, then
   a. Return OrdinaryDefineOwnProperty(A, "length", PropertyDescriptor{[[Writable]]: false}).
      This call will always return true.
18. Return true.

NOTE In steps 3 and 4, if Desc.[[Value]] is an object then its `valueOf` method is called twice. This is legacy behaviour that was specified with this effect starting with the 2nd Edition of this specification.

### 9.4.3 String Exotic Objects

A String object is an exotic object that encapsulates a String value and exposes virtual integer indexed data properties corresponding to the individual code unit elements of the String value. 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.

String exotic objects have the same internal slots as ordinary objects. They also have a `[[StringData]]` internal slot.

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

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

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

1. Assert: IsPropertyKey(P) is true.
2. Let desc be OrdinaryGetOwnProperty(S, P).
3. If desc is not undefined, return desc.
4. Return ! StringGetOwnProperty(S, P).

### 9.4.3.2 [[DefineOwnProperty]] (P, Desc)

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

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).

### 9.4.3.3 [[OwnPropertyKeys]] ()

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

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

### 9.4.3.4 StringCreate (value, prototype)

The abstract operation StringCreate with arguments value and prototype is used to specify the creation of new String exotic objects. It performs the following steps:
1. Assert: \( \text{Type(value)} \) is String.
2. Let \( S \) be a newly created String exotic object.
3. Set \( S[[\text{StringData}]] \) to \( \text{value} \).
4. Set \( S \)'s essential internal methods to the default ordinary object definitions specified in 9.1.
5. Set \( S[[\text{GetOwnProperty}]] \) as specified in 9.4.3.1.
6. Set \( S[[\text{DefineOwnProperty}]] \) as specified in 9.4.3.2.
7. Set \( S[[\text{OwnPropertyKeys}]] \) as specified in 9.4.3.3.
8. Set \( S[[\text{Prototype}]] \) to \( \text{prototype} \).
9. Set \( S[[\text{Extensible}]] \) to \( \text{true} \).
10. Let \( \text{length} \) be the number of code unit elements in \( \text{value} \).
11. Perform \( \text{DefinePropertyOrThrow}(S, \"\text{length}\", \text{PropertyDescriptor}[[[\text{Value}]]: \text{length}, [[\text{Writable}]]: \text{false}, [[\text{Enumerable}]]: \text{false}, [[\text{Configurable}]]: \text{false}) \).
12. Return \( S \).

### 9.4.3.5 StringGetOwnProperty \((S, P)\)

The abstract operation StringGetOwnProperty called with arguments \( S \) and \( P \) performs the following steps:

1. Assert: \( S \) is an Object that has a \([\text{StringData}]\) internal slot.
2. Assert: \( \text{IsPropertyKey}(P) \) is \( \text{true} \).
3. If \( \text{Type}(P) \) is not String, return \( \text{undefined} \).
4. Let \( \text{index} \) be \( \text{CanonicalNumericIndexString}(P) \).
5. If \( \text{index} \) is \( \text{undefined} \), return \( \text{undefined} \).
6. If \( \text{IsInteger}(\text{index}) \) is \( \text{false} \), return \( \text{undefined} \).
7. If \( \text{index} = -0 \), return \( \text{undefined} \).
8. Let \( \text{str} \) be the String value of \( S[[\text{StringData}]] \).
9. Let \( \text{len} \) be the number of elements in \( \text{str} \).
10. If \( \text{index} < 0 \) or \( \text{len} \leq \text{index} \), return \( \text{undefined} \).
11. Let \( \text{resultStr} \) be a String value of length 1, containing one code unit from \( \text{str} \), specifically the code unit at index \( \text{index} \).
12. Return a PropertyDescriptor\([[\text{Value}]]: \text{resultStr}, [[\text{Writable}]]: \text{false}, [[\text{Enumerable}]]: \text{true}, [[\text{Configurable}]]: \text{false}] \).

### 9.4.4 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.
Arguments exotic objects have the same internal slots as ordinary objects. They also have a 
[[ParameterMap]] internal slot. Ordinary arguments objects also have a [[ParameterMap]] internal slot 
whose value is always undefined. For ordinary argument objects the [[ParameterMap]] internal slot is only 
used by `Object.prototype.toString` (19.1.3.6) to identify them as such.

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

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

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

```
NOTE 3
Arguments objects for strict functions define a non-configurable accessor property named 
"callee" which throws a TypeError exception on access. The "callee" property has a 
more specific meaning for non-strict functions. The strict mode definition of this property 
exists to ensure that it is not defined in any other manner by conforming ECMAScript 
implementations.
```

```
NOTE 4
ECMAScript implementations of arguments objects have historically contained an accessor 
property named "caller". Prior to ECMAScript 2017, this specification included the 
definition of a throwing "caller" property on arguments objects for strict functions. 
Since implementations do not contain this extension any longer, ECMAScript 2017 
dropped the requirement for a throwing "caller" accessor.
```

9.4.4.1  [[GetOwnProperty]] ( P )

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

1. Let `args` be the arguments object.
2. Let desc be `OrdinaryGetOwnProperty(args, P)`.
3. If desc is `undefined`, return desc.
4. Let map be `args.[[ParameterMap]]`.
5. Let isMapped be `! HasOwnProperty(map, P)`.
6. If isMapped is `true`, then
   a. Set desc.[[Value]] to `Get(map, P)`.
7. Return desc.

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

The `[[DefineOwnProperty]]` internal method of an arguments exotic object when called with a property key P and Property Descriptor Desc performs the following steps:

1. Let args be the arguments object.
2. Let map be `args.[[ParameterMap]]`.
3. Let isMapped be `HasOwnProperty(map, P)`.
4. Let newArgDesc be Desc.
5. 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)`.
7. If allowed is `false`, return `false`.
8. 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).
9. Return `true`.

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

The `[[Get]]` internal method of an arguments exotic object when called with a property key P and ECMAScript language value Receiver performs the following steps:

1. Let args be the arguments object.
2. Let \( map \) be \( \text{args}[[\text{ParameterMap}]]. \)
3. Let \( \text{isMapped} \) be \( \text{! HasOwnProperty}(map, P) \).
4. If \( \text{isMapped} \) is \( \text{false} \), then
   a. Return \( \text{? OrdinaryGet}(\text{args}, P, \text{Receiver}) \).
5. Else \( map \) contains a formal parameter mapping for \( P \),
   a. Return \( \text{Get}(map, P) \).

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

The \([\text{Set}]\) internal method of an arguments exotic object when called with property key \( P \), value \( V \), and \( \text{ECMAScript language value Receiver} \) performs the following steps:

1. Let \( \text{args} \) be the arguments object.
2. If \( \text{SameValue}(\text{args}, \text{Receiver}) \) is \( \text{false} \), then
   a. Let \( \text{isMapped} \) be \( \text{false} \).
3. Else,
   a. Let \( map \) be \( \text{args}[[\text{ParameterMap}]] \).
   b. Let \( \text{isMapped} \) be \( \text{! HasOwnProperty}(map, P) \).
4. If \( \text{isMapped} \) is \( \text{true} \), then
   a. Let \( \text{setStatus} \) be \( \text{Set}(map, P, V, \text{false}) \).
   b. Assert: \( \text{setStatus} \) is \( \text{true} \) because formal parameters mapped by argument objects are always writable.
5. Return \( \text{? OrdinarySet}(\text{args}, P, V, \text{Receiver}) \).

### 9.4.4.5 \([\text{Delete}]\) \((P)\)

The \([\text{Delete}]\) internal method of an arguments exotic object when called with a property key \( P \) performs the following steps:

1. Let \( \text{args} \) be the arguments object.
2. Let \( map \) be \( \text{args}[[\text{ParameterMap}]] \).
3. Let \( \text{isMapped} \) be \( \text{! HasOwnProperty}(map, P) \).
4. Let \( \text{result} \) be \( \text{? OrdinaryDelete}(\text{args}, P) \).
5. If \( \text{result} \) is \( \text{true} \) and \( \text{isMapped} \) is \( \text{true} \), then
   a. Call \( map.[[\text{Delete}])(P) \).
6. Return \( \text{result} \).

### 9.4.4.6 CreateUnmappedArgumentsObject (argumentsList)

The abstract operation CreateUnmappedArgumentsObject called with an argument \( \text{argumentsList} \) performs the following steps:
1. Let `len` be the number of elements in `argumentsList`.
2. Let `obj` be `ObjectCreate(%ObjectPrototype%, « [[ParameterMap]] »)`.  
3. Set `obj.«[ParameterMap]»` to `undefined`.
4. Perform `DefinePropertyOrThrow(obj, "length", PropertyDescriptor{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).`
5. Let `index` be 0.
6. Repeat, while `index < len`,
   a. Let `val` be `argumentsList[index]`.
   b. Perform `CreateDataProperty(obj, ! ToString(index), val)`.
   c. Let `index` be `index + 1`.
7. Perform `! DefinePropertyOrThrow(obj, @@iterator, PropertyDescriptor {[[Value]]: %ArrayProto_values%, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true})`.
8. Perform `! DefinePropertyOrThrow(obj, "callee", PropertyDescriptor {[[Get]]: %ThrowTypeError%, [[Set]]: %ThrowTypeError%, [[Enumerable]]: false, [[Configurable]]: false})`.

### 9.4.4.7 CreateMappedArgumentsObject ( func, formals, argumentsList, env )

The abstract operation CreateMappedArgumentsObject is called with object `func`, Parse Node `formals`, List `argumentsList`, and Environment Record `env`. The following steps are performed:

1. Assert: `formals` does not contain a rest parameter, any binding patterns, or any initializers. It may contain duplicate identifiers.
2. Let `len` be the number of elements in `argumentsList`.
3. Let `obj` be a newly created arguments exotic object with a [[ParameterMap]] internal slot.
4. Set `obj.«[ParameterMap]»` as specified in 9.4.4.1.
5. Set `obj.«[DefineOwnProperty]»` as specified in 9.4.4.2.
6. Set `obj.«[Get]»` as specified in 9.4.4.3.
7. Set `obj.«[Set]»` as specified in 9.4.4.4.
8. Set `obj.«[Delete]»` as specified in 9.4.4.5.
9. Set the remainder of `obj`'s essential internal methods to the default ordinary object definitions specified in 9.1.
10. Set `obj.«[Prototype]»` to `%ObjectPrototype%`.
11. Set `obj.«[Extensible]»` to `true`.
12. Let `map` be `ObjectCreate(null)`.
13. Set `obj.«[ParameterMap]»` to `map`.
14. Let `parameterNames` be the BoundNames of `formals`.
15. Let `numberOfParameters` be the number of elements in `parameterNames`.
16. Let `index` be 0.
17. Repeat, while `index < len`,
   a. Let `val` be `argumentsList[index]`.
b. Perform \( \text{CreateDataProperty}(obj, !\text{ToString}(index), val) \).

c. Let \( index \) be \( index + 1 \).

18. Perform \( \text{DefinePropertyOrThrow}(obj, "length", \text{PropertyDescriptor}([[\text{Value}]: len, [[\text{Writable}]: true, [[\text{Enumerable}]: false, [[\text{Configurable}]: true}}]).

19. Let \( mappedNames \) be a new empty \( \text{List} \).

20. Let \( index \) be \( \text{numberOfParameters} - 1 \).

21. Repeat, while \( index \geq 0 \),

a. Let \( name \) be \( \text{parameterNames}[index] \).

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 \( \text{MakeArgGetter}(name, env) \).

2. Let \( p \) be \( \text{MakeArgSetter}(name, env) \).

3. Perform \( \text{map}([[\text{DefineOwnProperty}]](!\text{ToString}(index), \text{PropertyDescriptor}([[\text{Set}]: p, [[\text{Get}]: g, [[\text{Enumerable}]: false, [[\text{Configurable}]: true}})).

c. Let \( index \) be \( index - 1 \).

22. Perform \( \text{DefinePropertyOrThrow}(obj, @@\text{iterator}, \text{PropertyDescriptor} {{[[\text{Value}]]: \%ArrayProto_values\%, [[\text{Writable}]: true, [[\text{Enumerable}]: false, [[\text{Configurable}]: true}})).

23. Perform \( \text{DefinePropertyOrThrow}(obj, "callee", \text{PropertyDescriptor} {{[[\text{Value}]]: func, [[\text{Writable}]]: true, [[\text{Enumerable}]]: false, [[\text{Configurable}]]: true}})).

24. Return \( obj \).

9.4.4.7.1 \textbf{MakeArgGetter ( name, env )}

The abstract operation \text{MakeArgGetter} called with String \( name \) and \text{Environment Record} \( env \) creates a built-in function object that when executed returns the value bound for \( name \) in \( env \). It performs the following steps:

1. Let \( realm \) be the current Realm Record.

2. Let \( steps \) be the steps of an ArgGetter function as specified below.

3. Let \( getter \) be \( \text{CreateBuiltInFunction}(realm, steps, \%FunctionPrototype\%, \langle [[\text{Name}}], [[\text{Env}]] \rangle \).

4. Set \( getter.[[\text{Name}]] \) to \( name \).

5. Set \( getter.[[\text{Env}]] \) to \( env \).

6. Return \( getter \).

An ArgGetter function is an anonymous built-in function with \( [[\text{Name}]] \) and \( [[\text{Env}]] \) internal slots. When an ArgGetter function \( f \) that expects no arguments is called it performs the following steps:

1. Let \( name \) be \( f.[[\text{Name}]] \).

2. Let \( env \) be \( f.[[\text{Env}]] \).

3. Return \( env.\text{GetBindingValue}(name, false) \).
NOTE ArgGetter functions are never directly accessible to ECMAScript code.

### 9.4.4.7.2 MakeArgSetter (name, env)

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

1. Let realm be the current Realm Record.
2. Let steps be the steps of an ArgSetter function as specified below.
3. Let setter be CreateBuiltinFunction(realm, steps, %FunctionPrototype%, « [[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 f is called with argument value it performs the following steps:

1. Let name be f.[[Name]].
2. Let env be f.[[Env]].
3. Return env.SetMutableBinding(name, value, false).

NOTE ArgSetter functions are never directly accessible to ECMAScript code.

### 9.4.5 Integer Indexed Exotic Objects

An Integer Indexed object is an exotic object that performs special handling of integer index property keys.

Integer Indexed exotic objects have the same internal slots as ordinary objects and additionally [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.

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

#### 9.4.5.1 [[GetOwnProperty]] (P)

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

1. Assert: IsPropertyKey(P) is true.
2. Assert: O is an Object that has a [[ViewedArrayBuffer]] internal slot.
3. If Type(P) is String, then
a. Let `numericIndex` be `CanonicalNumericIndexString(P)`.
b. If `numericIndex` is not `undefined`, then
   i. Let `value` be `IntegerIndexedElementGet(O, numericIndex)`.
   ii. If `value` is `undefined`, return `undefined`.
      iii. Return a `PropertyDescriptor` with `[[Value]]: value, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: false`.
4. Return `OrdinaryGetOwnProperty(O, P)`.

### 9.4.5.2 [[HasProperty]](P)

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

1. Assert: `IsPropertyKey(P)` is `true`.
2. Assert: `O` is an Object that has a `[[ViewedArrayBuffer]]` internal slot.
3. If `Type(P)` is String, then
   a. Let `numericIndex` be `CanonicalNumericIndexString(P)`.
   b. If `numericIndex` is not `undefined`, then
      i. Let `buffer` be `O.[[ViewedArrayBuffer]]`.
      ii. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
      iii. If `IsInteger(numericIndex)` is `false`, return `false`.
      iv. If `numericIndex` = `-0`, return `false`.
      v. If `numericIndex` < 0, return `false`.
      vi. If `numericIndex` ≥ `O.[[ArrayLength]]`, return `false`.
      vii. Return `true`.

### 9.4.5.3 [[DefineOwnProperty]](P, Desc)

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

1. Assert: `IsPropertyKey(P)` is `true`.
2. Assert: `O` is an Object that has a `[[ViewedArrayBuffer]]` internal slot.
3. If `Type(P)` is String, then
   a. Let `numericIndex` be `CanonicalNumericIndexString(P)`.
   b. If `numericIndex` is not `undefined`, then
      i. If `IsInteger(numericIndex)` is `false`, return `false`.
      ii. If `numericIndex` = `-0`, return `false`.
      iii. If `numericIndex` < 0, return `false`.
      iv. Let `length` be `O.[[ArrayLength]]`.
v. If \( \text{numericIndex} \geq \text{length} \), return \textbf{false}.

vi. If \( \text{IsAccessorDescriptor}(\text{Desc}) \) is \textbf{true}, return \textbf{false}.

vii. If \( \text{Desc} \) has a [[Configurable]] field and if \( \text{Desc}.[[\text{Configurable}]] \) is \textbf{true}, return \textbf{false}.

viii. If \( \text{Desc} \) has an [[Enumerable]] field and if \( \text{Desc}.[[\text{Enumerable}]] \) is \textbf{false}, return \textbf{false}.

ix. If \( \text{Desc} \) has a [[Writable]] field and if \( \text{Desc}.[[\text{Writable}]] \) is \textbf{false}, return \textbf{false}.

x. If \( \text{Desc} \) has a [[Value]] field, then
   1. Let \( \text{value} \) be \( \text{Desc}.[[\text{Value}]] \).
   2. Return \( \text{? IntegerIndexedElementSet}(O, \text{numericIndex}, \text{value}) \).

xi. Return \textbf{true}.

4. Return \( \text{! OrdinaryDefineOwnProperty}(O, P, \text{Desc}) \).

### 9.4.5.4 [[Get]] (\( P, \text{Receiver} \))

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

1. Assert: \( \text{IsPropertyKey}(P) \) is \textbf{true}.

2. If \( \text{Type}(P) \) is String, then
   a. Let \( \text{numericIndex} \) be \( \text{! CanonicalNumericIndexString}(P) \).
   b. If \( \text{numericIndex} \) is not \textbf{undefined}, then
      i. Return \( \text{? IntegerIndexedElementGet}(O, \text{numericIndex}) \).

3. Return \( \text{? OrdinaryGet}(O, P, \text{Receiver}) \).

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

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

1. Assert: \( \text{IsPropertyKey}(P) \) is \textbf{true}.

2. If \( \text{Type}(P) \) is String, then
   a. Let \( \text{numericIndex} \) be \( \text{! CanonicalNumericIndexString}(P) \).
   b. If \( \text{numericIndex} \) is not \textbf{undefined}, then
      i. Return \( \text{? IntegerIndexedElementSet}(O, \text{numericIndex}, V) \).

3. Return \( \text{? OrdinarySet}(O, P, V, \text{Receiver}) \).

### 9.4.5.6 [[OwnPropertyKeys]] ()

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

1. Let \( \text{keys} \) be a new empty \texttt{List}.
2. Assert: \( O \) is an Object that has \([[[ViewedArrayBuffer]]], [[[ArrayLength]]], [[[ByteOffset]]], and \([[[TypedArrayName]]]\) internal slots.
3. Let \( len \) be \( O.[[[ArrayLength]]]. \)
4. For each integer \( i \) starting with 0 such that \( i < len \), in ascending order, do
   a. Add \( \text{!ToString}(i) \) as the last element of \( \text{keys} \).
5. 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 \( \text{keys} \).
6. 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 \( \text{keys} \).
7. Return \( \text{keys} \).

9.4.5.7 IntegerIndexedObjectCreate (prototype, internalSlotsList)

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

1. Assert: \( \text{internalSlotsList} \) contains the names \([[[ViewedArrayBuffer]]], [[[ArrayLength]]], [[[ByteOffset]]], and \([[[TypedArrayName]]]\).
2. Let \( A \) be a newly created object with an internal slot for each name in \( \text{internalSlotsList} \).
3. Set \( A \)'s essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set \( A.[[[GetOwnProperty]]] \) as specified in 9.4.5.1.
5. Set \( A.[[[HasProperty]]] \) as specified in 9.4.5.2.
6. Set \( A.[[[DefineOwnProperty]]] \) as specified in 9.4.5.3.
7. Set \( A.[[[Get]]] \) as specified in 9.4.5.4.
8. Set \( A.[[[Set]]] \) as specified in 9.4.5.5.
9. Set \( A.[[[OwnPropertyKeys]]] \) as specified in 9.4.5.6.
10. Set \( A.[[[Prototype]]] \) to \( \text{prototype} \).
11. Set \( A.[[[Extensible]]] \) to \( \text{true} \).
12. Return \( A \).

9.4.5.8 IntegerIndexedElementGet (0, index)

The abstract operation IntegerIndexedElementGet with arguments \( 0 \) and \( index \) performs the following steps:

1. Assert: \( \text{Type}(index) \) is Number.
2. Assert: $O$ is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let $buffer$ be $O$.[[ViewedArrayBuffer]].
4. If IsDetachedBuffer($buffer$) is true, throw a TypeError exception.
5. If IsInteger($index$) is false, return undefined.
6. If $index = -0$, return undefined.
7. Let $length$ be $O$.[[ArrayLength]].
8. If $index < 0$ or $index \geq length$, return undefined.
9. Let $offset$ be $O$.[[ByteOffset]].
10. Let $arrayTypeName$ be the String value of $O$.[[TypedArrayName]].
11. Let $elementSize$ be the Number value of the Element Size value specified in Table 50 for $arrayTypeName$.
12. Let $indexedPosition$ be $(index \times elementSize) + offset$.
13. Let $elementType$ be the String value of the Element Type value in Table 50 for $arrayTypeName$.
14. Return GetValueFromBuffer($buffer$, $indexedPosition$, $elementType$, true, "Unordered").

9.4.5.9 IntegerIndexedElementSet ($O$, $index$, $value$)

The abstract operation IntegerIndexedElementSet with arguments $O$, $index$, and $value$ performs the following steps:

1. Assert: Type($index$) is Number.
2. Assert: $O$ is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let $numValue$ be ? ToNumber($value$).
4. Let $buffer$ be $O$.[[ViewedArrayBuffer]].
5. If IsDetachedBuffer($buffer$) is true, throw a TypeError exception.
6. If IsInteger($index$) is false, return false.
7. If $index = -0$, return false.
8. Let $length$ be $O$.[[ArrayLength]].
9. If $index < 0$ or $index \geq length$, return false.
10. Let $offset$ be $O$.[[ByteOffset]].
11. Let $arrayTypeName$ be the String value of $O$.[[TypedArrayName]].
12. Let $elementSize$ be the Number value of the Element Size value specified in Table 50 for $arrayTypeName$.
13. Let $indexedPosition$ be $(index \times elementSize) + offset$.
14. Let $elementType$ be the String value of the Element Type value in Table 50 for $arrayTypeName$.
15. Perform SetValueInBuffer($buffer$, $indexedPosition$, $elementType$, $numValue$, true, "Unordered").
16. Return true.
9.4.6 Module Namespace Exotic Objects

A module namespace object is an exotic object that exposes the bindings exported from an ECMAScript Module (See 15.2.3). 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 objects are not extensible.

Module namespace objects have the internal slots defined in Table 29.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Module]]</td>
<td>Module Record</td>
<td>The Module Record whose exports this namespace exposes.</td>
</tr>
<tr>
<td>[[Exports]]</td>
<td>List of String</td>
<td>A List containing the String values of the exported names exposed as own properties of this object. The list is ordered as if an Array of those String values had been sorted using <code>Array.prototype.sort</code> using undefined as <code>comparefn</code>.</td>
</tr>
<tr>
<td>[[Prototype]]</td>
<td>Null</td>
<td>This slot always contains the value <code>null</code> (see 9.4.6.1).</td>
</tr>
</tbody>
</table>

Module namespace exotic objects provide alternative definitions for all of the internal methods except `[[GetPrototypeOf]]`, which behaves as defined in 9.1.1.

9.4.6.1 `[[SetPrototypeOf]] ( V )`

When the `[[SetPrototypeOf]]` internal method of a module namespace exotic object `O` is called with argument `V`, the following steps are taken:

1. Return `SetImmutablePrototype(O, V)`.

9.4.6.2 `[[IsExtensible]] ()`

When the `[[IsExtensible]]` internal method of a module namespace exotic object `O` is called, the following steps are taken:

1. Return `false`. 
9.4.6.3 [[PreventExtensions]] ()

When the [[PreventExtensions]] internal method of a module namespace exotic object *O* is called, the following steps are taken:

1. Return `true`.

9.4.6.4 [[GetOwnProperty]] ( *P*)

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

1. If `Type(P)` is `Symbol`, return OrdinaryGetOwnProperty(*O*, *P*).
2. Let `exports` be *O*.[[Exports]].
3. If *P* is not an element of `exports`, return `undefined`.
4. Let `value` be ? *O*.[[Get]](*P*, *O*).
5. Return PropertyDescriptor{[[Value]]: `value`, [[Writable]]: `true`, [[Enumerable]]: `true`, [[Configurable]]: `false`}.

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

When the [[DefineOwnProperty]] internal method of a module namespace exotic object *O* is called with property key *P* and Property Descriptor `Desc`, the following steps are taken:

1. Return `false`.

9.4.6.6 [[HasProperty]] ( *P*)

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

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`.

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

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

1. Assert: IsPropertyKey(*P*) is `true`.
2. If `Type(P)` is `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]].
7. Assert: binding is neither null nor "ambiguous".
8. Let targetModule be binding.[[Module]].
9. Assert: targetModule is not undefined.
10. Let targetEnv be targetModule.[[Environment]].
11. If targetEnv is undefined, throw a ReferenceError exception.
12. Let targetEnvRec be targetEnv's EnvironmentRecord.

NOTE ResolveExport is idempotent and side-effect free. An implementation might choose to pre-compute or cache the ResolveExport results for the [[Exports]] of each module namespace exotic object.

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

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

1. Return false.

9.4.6.9 [[Delete]] (P)

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

1. Assert: IsPropertyKey(P) is true.
2. Let exports be O.[[Exports]].
3. If P is an element of exports, return false.
4. Return true.

9.4.6.10 [[OwnPropertyKeys]] ( )

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

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

### 9.4.6.11 ModuleNamespaceCreate (module, exports)

The abstract operation ModuleNamespaceCreate with arguments `module`, and `exports` is used to specify the creation of new module namespace exotic objects. It performs the following steps:

1. Assert: `module` is a Module Record.
2. Assert: `module` is undefined.
3. Assert: `exports` is a List of String values.
4. Let `M` be a newly created object.
5. Set `M`'s essential internal methods to the definitions specified in 9.4.6.
6. Set `M`[[Module]] to `module`.
7. Let `sortedExports` be a new List containing the same values as the list `exports` where the values are ordered as if an Array of the same values had been sorted using `Array.prototype.sort` using undefined as comparefn.
8. Set `M`[[Exports]] to `sortedExports`.
9. Create own properties of `M` corresponding to the definitions in 26.3.
10. Set `module`[[Namespace]] to `M`.
11. Return `M`.

### 9.4.7 Immutable Prototype Exotic Objects

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

Immutable prototype exotic objects have the same internal slots as ordinary objects. They are exotic only in the following internal methods. All other internal methods of immutable prototype exotic objects that are not explicitly defined below are instead defined as in ordinary objects.

#### 9.4.7.1 [[SetPrototypeOf]] (V)

When the [[SetPrototypeOf]] internal method of an immutable prototype exotic object `O` is called with argument `V`, the following steps are taken:


#### 9.4.7.2 SetImmutablePrototype (O, V)

When the SetImmutablePrototype abstract operation is called with arguments `O` and `V`, the following steps are taken:
1. Assert: Either $\text{Type}(V)$ is Object or $\text{Type}(V)$ is Null.
2. Let $\text{current}$ be $O.\text{[[GetPrototypeOf]]}()$.
3. If $\text{SameValue}(V, \text{current})$ is true, return true.
4. Return false.

9.5 Proxy Object Internal Methods and Internal Slots

A proxy object is an exotic object whose essential internal methods are partially implemented using ECMAScript code. Every proxy objects has an internal slot called $\text{[[ProxyHandler]]}$. The value of $\text{[[ProxyHandler]]}$ is an object, called the proxy's handler object, or null. Methods (see Table 30) 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 $\text{[[ProxyTarget]]}$ whose value is either an object or the null value. This object is called the proxy's target object.

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

### 9.5.1 [[GetPrototypeOf]] ( )

When the [[GetPrototypeOf]] internal method of a Proxy exotic object O is called, the following steps are taken:

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, "getPrototypeOf").
6. If trap is undefined, then
   a. Return ? target.[[GetPrototypeOf]]().
7. Let handlerProto be ? Call(trap, handler, « target »).
8. If Type(handlerProto) is neither Object nor Null, throw a TypeError exception.
10. If extensibleTarget is true, return handlerProto.
11. Let targetProto be ? target.[[GetPrototypeOf]]().
12. If SameValue(handlerProto, targetProto) is false, throw a TypeError exception.

---

**NOTE**

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

### 9.5.2 `[[SetPrototypeOf]] (V)

When the `[[SetPrototypeOf]]` internal method of a Proxy exotic object `O` is called with argument `V`, the following steps are taken:

1. Assert: Either `Type(V)` is Object or `Type(V)` is `null`.
2. 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
   a. Return `target.[[SetPrototypeOf]](V)`.
8. Let `booleanTrapResult` be `ToBoolean(? Call(trap, handler, « target, V »))`.
9. If `booleanTrapResult` is `false`, return `false`.
10. Let `extensibleTarget` be `? IsExtensible(target)`.
11. If `extensibleTarget` is `true`, return `true`.
13. If `SameValue(V, targetProto)` is `false`, throw a `TypeError` exception.

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

- The result of `[[SetPrototypeOf]]` is a Boolean value.
- If the target object is not extensible, the argument value must be the same as the result of `[[GetPrototypeOf]]` applied to target object.

### 9.5.3 `[[IsExtensible]] ()

When the `[[IsExtensible]]` internal method of a Proxy exotic object `O` is called, the following steps are taken:

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]]`. 

NOTE `[[IsExtensible]]` can be used to check if the target object is extensible.
5. Let \( \text{trap} \) be \( \text{getMethod}(\text{handler}, "\text{isExtensible}"). 
6. If \( \text{trap} \) is undefined, then 
   a. Return \( \text{target}[[\text{IsExtensible}]]() \).
7. Let boolean\( \text{trapResult} \) be ToBoolean(\( \text{Call}(\text{trap}, \text{handler}, \text{"target"}) \)).
8. Let \( \text{targetResult} \) be \( \text{target}[[\text{IsExtensible}]]() \).
9. If \( \text{SameValue}(\text{boolean\( \text{trapResult}, \text{targetResult} \))) \) is false, throw a \text{TypeError} exception.
10. Return \( \text{boolean\( \text{trapResult} \))}.

**NOTE** 

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

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

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

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

1. Let \( \text{handler} \) be \( O.[[\text{ProxyHandler}]] \).
2. If \( \text{handler} \) is null, throw a \text{TypeError} exception.
3. Assert: \( \text{Type}(\text{handler}) \) is Object.
4. Let \( \text{target} \) be \( O.[[\text{ProxyTarget}]] \).
5. Let \( \text{trap} \) be \( \text{getMethod}(\text{handler}, "\text{preventExtensions}"). 
6. If \( \text{trap} \) is undefined, then 
   a. Return \( \text{target}[[\text{PreventExtensions}]]() \).
7. Let boolean\( \text{trapResult} \) be ToBoolean(\( \text{Call}(\text{trap}, \text{handler}, \text{"target"}) \)).
8. If \( \text{boolean\( \text{trapResult} \))) \) is true, then 
   a. Let targetIs\( \text{Extensible} \) be \( \text{target}[[\text{IsExtensible}]]() \).
   b. If targetIs\( \text{Extensible} \) is true, throw a \text{TypeError} exception.
9. Return \( \text{boolean\( \text{trapResult} \))}.

**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 only returns true if 
  \( [\text{IsExtensible}] \) applied to the proxy object’s target object is false.

### 9.5.5 \( [\text{GetOwnProperty}]\( P \)\)
When the `[[GetOwnProperty]]` internal method of a Proxy exotic object `O` is called with property key `P`, the following steps are taken:

1. Assert: `IsPropertyKey(P)` is true.
2. Let `handler` be `O.[[ProxyHandler]]`.
3. If `handler` is null, throw a `TypeError` exception.
4. Assert: `Type(handler)` is Object.
5. Let `target` be `O.[[ProxyTarget]]`.
6. Let `trap` be `getMethod(handler, "getOwnPropertyDescriptor")`.
7. If `trap` is undefined, then
   a. Return `target. [[GetOwnProperty]](P)`.
8. Let `trapResultObj` be `Call(trap, handler, « target, P »)`.
9. If `Type(trapResultObj)` is neither Object nor Undefined, throw a `TypeError` exception.
10. Let `targetDesc` be `target. [[GetOwnProperty]](P)`.
11. If `trapResultObj` is undefined, then
    a. If `targetDesc` is undefined, return undefined.
    b. If `targetDesc. [[Configurable]]` is false, throw a `TypeError` exception.
    c. Let `extensibleTarget` be `IsExtensible(target)`.
    d. Assert: `Type(extensibleTarget)` is Boolean.
    e. If `extensibleTarget` is false, throw a `TypeError` exception.
    f. 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.
18. Return `resultDesc`.

**NOTE**

`[[GetOwnProperty]]` for proxy objects enforces the following invariants:

- The result of `[[GetOwnProperty]]` must be either an Object or `undefined`.
- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as an own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exist as an own property of the target object and the target object is not extensible.
A property cannot be reported as non-configurable, if it does not exist as an own property of the target object or if it exists as a configurable own property of the target object.

### 9.5.6 [[DefineOwnProperty]] (P, Desc)

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

1. Assert: IsPropertyKey(P) is true.
2. 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 targetDesc is not undefined,
   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.
17. Return true.

### NOTE

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

- The result of [[DefineOwnProperty]] is a Boolean value.
- A property cannot be added, if the target object is not extensible.
A property cannot be non-configurable, unless there exists a corresponding non-configurable own property of the target object. If a property has a corresponding target object property then applying the Property Descriptor of the property to the target object using [[DefineOwnProperty]] will not throw an exception.

### 9.5.7 [[HasProperty]] (P)

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

1. Assert: IsPropertyKey\((P)\) is true.
2. Let \( handler \) be \( O.\text{[[ProxyHandler]]} \).
3. If \( handler \) is null, throw a TypeError exception.
4. Assert: Type\((handler)\) is Object.
5. Let \( target \) be \( O.\text{[[ProxyTarget]]} \).
6. Let \( trap \) be \( ? \text{GetMethod} (\text{handler}, \"has\") \).
7. If \( trap \) is undefined, then
   a. Return \( ? \text{target}.\text{[[HasProperty]]}(P) \).
8. Let \( booleanTrapResult \) be \( \text{ToBoolean}(? \text{Call}(\text{trap}, \text{handler}, \text{« target, P »})) \).
9. If \( booleanTrapResult \) is false, then
   a. Let \( targetDesc \) be \( ? \text{target}.\text{[[GetOwnProperty]]}(P) \).
   b. If \( targetDesc \) is not undefined, then
      i. If \( targetDesc.\text{[[Configurable]]} \) is false, throw a TypeError exception.
      ii. Let \( extensibleTarget \) be ? IsExtensible\((target)\).
      iii. If \( extensibleTarget \) is false, throw a TypeError exception.
10. Return \( booleanTrapResult \).

**NOTE**

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

- The result of [[HasProperty]] is a Boolean value.
- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as an own property of the target object and the target object is not extensible.

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

When the [[Get]] internal method of a Proxy exotic object \( O \) is called with property key \( P \) and ECMAScript language value Receiver, the following steps are taken:
1. Assert: `IsPropertyKey(P)` is true.
2. 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 `trapResult` be ? `Call(trap, handler, « target, P, Receiver »).
10. If `targetDesc` is not `undefined`, then
   a. If `IsDataDescriptor(targetDesc)` is true and `targetDesc.[[Configurable]]` is false 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.[[Configurable]]` is false 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.

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

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

1. Assert: `IsPropertyKey(P)` is true.
2. 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
a. Return \( \texttt{target}[[\texttt{Set}]](P, V, \texttt{Receiver}) \).

8. Let \( \texttt{booleanTrapResult} \) be \( \texttt{ToBoolean(\texttt{Call(trap, handler, « target, P, V, Receiver »})}) \).

9. If \( \texttt{booleanTrapResult} \) is \texttt{false}, return \texttt{false}.

10. Let \( \texttt{targetDesc} \) be \( \texttt{target}[[\texttt{GetOwnProperty}]](P) \).

11. If \( \texttt{targetDesc} \) is not \texttt{undefined}, then
    
    a. If \( \texttt{IsDataDescriptor(\texttt{targetDesc})} \) is \texttt{true} and \( \texttt{targetDesc}[[\texttt{Configurable}]] \) is \texttt{false} and \( \texttt{targetDesc}[[\texttt{Writable}]] \) is \texttt{false}, then
        i. If \( \texttt{SameValue(V, targetDesc[[Value]])} \) is \texttt{false}, throw a \texttt{TypeError} exception.
    
    b. If \( \texttt{IsAccessorDescriptor(\texttt{targetDesc})} \) is \texttt{true} and \( \texttt{targetDesc}[[\texttt{Configurable}]] \) is \texttt{false}, then
        i. If \( \texttt{targetDesc}[[\texttt{Set}]] \) is \texttt{undefined}, throw a \texttt{TypeError} exception.

12. Return \texttt{true}.

**NOTE**

\( [[\texttt{Set}]] \) for proxy objects enforces the following invariants:

- The result of \( [[\texttt{Set}]] \) 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 \texttt{undefined} as its \( [[\texttt{Set}]] \) attribute.

### 9.5.10 \( [[\texttt{Delete}]](P) \)

When the \( [[\texttt{Delete}]] \) internal method of a Proxy exotic object \( O \) is called with property key \( P \), the following steps are taken:

1. Assert: \( \texttt{IsPropertyKey(P)} \) is \texttt{true}.
2. Let \( \texttt{handler} \) be \( O[[\texttt{ProxyHandler}]] \).
3. If \( \texttt{handler} \) is \texttt{null}, throw a \texttt{TypeError} exception.
4. Assert: \( \texttt{Type(handler)} \) is \texttt{Object}.
5. Let \( \texttt{target} \) be \( O[[\texttt{ProxyTarget}]] \).
6. Let \( \texttt{trap} \) be \( \texttt{GetMethod(handler, "deleteProperty")} \).
7. If \( \texttt{trap} \) is \texttt{undefined}, then
   a. Return \( \texttt{target}[[\texttt{Delete}]](P) \).
8. Let \( \texttt{booleanTrapResult} \) be \( \texttt{ToBoolean(\texttt{Call(trap, handler, « target, P »})}) \).
9. If \( \texttt{booleanTrapResult} \) is \texttt{false}, return \texttt{false}.
10. Let \( \texttt{targetDesc} \) be \( \texttt{target}[[\texttt{GetOwnProperty}]](P) \).
11. If \( \texttt{targetDesc} \) is \texttt{undefined}, return \texttt{true}.
12. If \( \texttt{targetDesc}[[\texttt{Configurable}]] \) is \texttt{false}, throw a \texttt{TypeError} exception.
13. Return \texttt{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.

9.5.11  [[OwnPropertyKeys]] ( )

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

1. Let \( handler \) be \( O.\text{[[ProxyHandler]]} \).
2. If \( handler \) is \text{null}, throw a \text{TypeError} exception.
3. Assert: \text{Type}(\text{handler}) is Object.
4. Let \( target \) be \( O.\text{[[ProxyTarget]]} \).
5. Let \( trap \) be ? \text{GetMethod}(\text{handler}, \text{"ownKeys"}).
6. If \( trap \) is \text{undefined}, then
   a. Return ? \text{target.[[OwnPropertyKeys]]}()
7. Let \( trapResultArray \) be ? \text{Call}(\text{trap}, \text{handler}, \text{« target »})
8. Let \( trapResult \) be ? \text{CreateListFromArrayLike}(\text{trapResultArray}, \text{« String, Symbol »})
9. Let \( extensibleTarget \) be ? \text{IsExtensible}(\text{target})
10. Let \( targetKeys \) be ? \text{target.[[OwnPropertyKeys]]}()
11. Assert: \( targetKeys \) is a List containing only String and Symbol values.
12. Let \( targetConfigurableKeys \) be a new empty List.
13. Let \( targetNonconfigurableKeys \) be a new empty List.
14. For each element \( key \) of \( targetKeys \), do
   a. Let \( desc \) be ? \text{target.[[GetProperty]]}(key).
   b. If \( desc \) is not \text{undefined} and \( desc.\text{[[Configurable]]} \) is \text{false}, then
      i. Append \( key \) as an element of \( targetNonconfigurableKeys \).
   c. Else,
      i. Append \( key \) as an element of \( targetConfigurableKeys \).
15. If \( extensibleTarget \) is \text{true} and \( targetNonconfigurableKeys \) is empty, then
    a. Return \( trapResult \).
16. Let \( uncheckedResultKeys \) be a new List which is a copy of \( trapResult \)
17. For each \( key \) that is an element of \( targetNonconfigurableKeys \), do
    a. If \( key \) is not an element of \( uncheckedResultKeys \), throw a \text{TypeError} exception.
    b. Remove all occurrences of \( key \) from \( uncheckedResultKeys \).
18. If \( extensibleTarget \) is \text{true}, return \( trapResult \).
19. For each \( key \) that is an element of \( targetConfigurableKeys \), do
    a. If \( key \) is not an element of \( uncheckedResultKeys \), throw a \text{TypeError} exception.
b. Remove all occurrences of \textit{key} from \textit{uncheckedResultKeys}.

20. If \textit{uncheckedResultKeys} is not empty, throw a \texttt{TypeError} exception.

21. Return \texttt{trapResult}.

\begin{verbatim}
NOTE [[OwnPropertyKeys]] for proxy objects enforces the following invariants:

- The result of [[OwnPropertyKeys]] is a \texttt{List}.
- The Type of each result \texttt{List} element is either String or Symbol.
- The result \texttt{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 \texttt{List} must contain all the keys of the own properties of the target object and no other values.
\end{verbatim}

### 9.5.12 \texttt{[[Call]]} ( \texttt{thisArgument, argumentsList} )

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

1. Let \texttt{handler} be \texttt{O[[ProxyHandler]]}.
2. If \texttt{handler} is \texttt{null}, throw a \texttt{TypeError} exception.
3. Assert: \texttt{Type(handler)} is Object.
4. Let \texttt{target} be \texttt{O[[ProxyTarget]]}.
5. Let \texttt{trap be ? GetMethod(handler, "apply")}.
6. If \texttt{trap} is \texttt{undefined}, then
   a. Return \texttt{? Call(target, thisArgument, argumentsList)}.
7. Let \texttt{argArray be CreateArrayFromList(argumentsList)}.
8. Return \texttt{? Call(trap, handler, « target, thisArgument, argArray »)}.

\begin{verbatim}
NOTE A Proxy exotic object only has a \texttt{[[Call]]} internal method if the initial value of its [[ProxyTarget]] internal slot is an object that has a \texttt{[[Call]]} internal method.
\end{verbatim}

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

The \texttt{[[Construct]]} internal method of a Proxy exotic object \texttt{O} is called with parameters \texttt{argumentsList} which is a possibly empty \texttt{List} of ECMAScript language values and \texttt{newTarget}. The following steps are taken:

1. Let \texttt{handler} be \texttt{O[[ProxyHandler]]}.
2. If \texttt{handler} is \texttt{null}, throw a \texttt{TypeError} exception.
3. Assert: \texttt{Type(handler)} is Object.
4. Let \texttt{target} be \texttt{O[[ProxyTarget]]}.
5. Let \texttt{trap be ? GetMethod(handler, "construct")}.
6. If trap is undefined, then
   a. Assert: target has a [[Construct]] internal method.
7. Let argArray be CreateArrayFromList(argumentsList).
8. Let newObj be ? Call(trap, handler, « target, argArray, newTarget »).
9. If Type(newObj) is not Object, throw a TypeError exception.
10. Return newObj.

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

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

9.5.14 ProxyCreate ( target, handler )

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

1. If Type(target) is not Object, throw a TypeError exception.
2. If target is a Proxy exotic object and target.[[ProxyHandler]] is null, throw a TypeError exception.
3. If Type(handler) is not Object, throw a TypeError exception.
4. If handler is a Proxy exotic object and handler.[[ProxyHandler]] is null, throw a TypeError
   exception.
5. Let P be a newly created object.
6. Set P's essential internal methods (except for [[Call]] and [[Construct]]) to the definitions specified in
   9.5.
7. If IsCallable(target) is true, then
   a. Set P.[[Call]] as specified in 9.5.12.
   b. If target has a [[Construct]] internal method, then
      i. Set P.[[Construct]] as specified in 9.5.13.
8. Set P.[[ProxyTarget]] to target.
9. Set P.[[ProxyHandler]] to handler.
10. Return P.

10 ECMAScript Language: Source Code
10.1 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.

10.1.1 Static Semantics: UTF16Encoding ( cp )

The UTF16Encoding of a numeric code point value, cp, is determined as follows:
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*, *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 9.2) 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*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.
- **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*, *MethodDefinition*, *ArrowFunction*, *AsyncArrowFunction*, *ClassDeclaration*, or *ClassExpression*.

**NOTE** Function code is generally provided as the bodies of Function Definitions (14.1), Arrow

---

1. Assert: \(0 \leq cp \leq 0x10FFFF\).
2. If \( cp \leq 0xFFFF\), return \( cp \).
3. Let \( cu1\) be \(\text{floor}((cp - 0 \times 10000) / 0x400) + 0xD800\).
4. Let \( cu2\) be \((cp - 0 \times 10000) \mod 0x400) + 0xDC00\).
5. Return the code unit sequence consisting of \( cu1\) followed by \( cu2\).
Function Definitions (14.2), Method Definitions (14.3), Generator Definitions (14.4), Async Function Definitions (14.6), and Async Arrow Functions (14.7). Function code is also derived from the arguments to the Function constructor (19.2.1.1), the GeneratorFunction constructor (25.2.1.1), and the AsyncFunction constructor (25.5.1.1).

10.2.1 Strict Mode Code

An ECMAScript Script syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. 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, MethodDefinition, ArrowFunction, or AsyncArrowFunction is contained in strict mode code or if the code that produces the value of the function’s [[ECMAScriptCode]] internal slot begins with a Directive Prologue that contains a Use Strict Directive.
- Function code that is supplied as the arguments to the built-in Function, Generator, and AsyncFunction constructors is strict mode code if the last argument is a String that when processed is a FunctionBody that begins with a Directive Prologue that contains a Use Strict Directive.

ECMAScript code that is not strict mode code is called non-strict code.

10.2.2 Non-ECMAScript Functions

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

11 ECMAScript Language: Lexical Grammar
The source text of an ECMAScript Script 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 11.9); in examples such as the following:

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

where the first non-whitespace, non-comment code point after a `LineTerminator` is U+002F (SOLIDUS) and the syntactic context allows division or division-assignment, no semicolon is inserted at the `LineTerminator`. That is, the above example is interpreted in the same way as:

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

**Syntax**

```
InputElementDiv  ::
    WhiteSpace
    LineTerminator
    Comment
    CommonToken
    DivPunctuator
    RightBracePunctuator

InputElementRegExp  ::
```
11.1 Unicode Format-Control Characters

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

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

U+200C (ZERO WIDTH NON-JOINER) and U+200D (ZERO WIDTH JOINER) are format-control characters that are used to make necessary distinctions when forming words or phrases in certain languages. In ECMAScript source text these code points may also be used in an IdentifierName 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 11.2).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarized in Table 31.

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Abbreviation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+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>

### 11.2 White Space

White space code points are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space code points may occur between any two tokens and at the start or end of input. White space code points may occur within a `StringLiteral`, a `RegularExpressionLiteral`, a `Template`, or a `TemplateSubstitutionTail` where they are considered significant code points forming part of a literal value. They may also occur within a `Comment`, but cannot appear within any other kind of token.

The ECMAScript white space code points are listed in Table 32.

<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 32, ECMAScript WhiteSpace intentionally excludes all code points that have the Unicode “White_Space” property but which are not classified in category “Space_Separator” ("Zs").

Syntax

WhiteSpace ::
   <TAB>
   <VT>
   <FF>
   <SP>
   <NBSP>
   <ZWNBSP>
   <USP>

11.3 Line Terminators

Like white space code points, line terminator code points are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space code points, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. Line terminators also affect the process of automatic semicolon insertion (11.9). A line terminator cannot occur within any token except a StringLiteral, Template, or TemplateSubstitutionTail. Line terminators may only occur within a StringLiteral token as part of a LineContinuation.

A line terminator can occur within a MultiLineComment 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 33.

<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+00DD</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>
</tbody>
</table>
Only the Unicode code points in Table 33 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 32. 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

\[
\text{LineTerminator} ::
\]
\[
<\text{LF}>
\]
\[
<\text{CR}>
\]
\[
<\text{LS}>
\]
\[
<\text{PS}>
\]

\[
\text{LineTerminatorSequence} ::
\]
\[
<\text{LF}>
\]
\[
<\text{CR}>[\text{lookahead} \neq \text{LF}]\]
\[
<\text{LS}>
\]
\[
<\text{PS}>
\]
\[
<\text{CR}><\text{LF}>
\]

11.4 Comments

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

Because a single-line comment can contain any Unicode code point except a LineTerminator code point, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all code points from the // marker to the end of the line. However, the LineTerminator at the end of the line is not considered to be part of the single-line comment; it is recognized separately by the lexical grammar and becomes part of the stream of input elements for the syntactic grammar. This point is very important, because it implies that the presence or absence of single-line comments does not affect the process of automatic semicolon insertion (see 11.9).

Comments behave like white space and are discarded except that, if a MultiLineComment contains a line terminator code point, then the entire comment is considered to be a LineTerminator for purposes of parsing by the syntactic grammar.

Syntax
Comment ::
    MultiLineComment
    SingleLineComment

MultiLineComment ::
    /* 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

11.5 Tokens

Syntax

CommonToken ::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral
    Template

NOTE The DivPunctuator, RegularExpressionLiteral, RightBracePunctuator, and
11.6 Names and Keywords

*IdentifierName* and *ReservedWord* are tokens that are interpreted according to the Default Identifier Syntax given in Unicode Standard Annex #31, Identifier and Pattern Syntax, with some small modifications. *ReservedWord* is an enumerated subset of *IdentifierName*. The syntactic grammar defines *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 *HexDigits* of the *UnicodeEscapeSequence* (see 11.8.4). The \ preceding the *UnicodeEscapeSequence* and the u and { } code units, if they appear, do not contribute code points to the *IdentifierName*. A *UnicodeEscapeSequence* cannot be used to put a code point into an *IdentifierName* that would otherwise be illegal. In other words, if a \ *UnicodeEscapeSequence* sequence were replaced by the *SourceCharacter* it contributes, the result must still be a valid *IdentifierName* that has the exact same sequence of *SourceCharacter* elements as the original *IdentifierName*. All interpretations of *IdentifierName* within this specification are based upon their actual code points regardless of whether or not an escape sequence was used to contribute any particular code point.

Two *IdentifierNames* that are canonically equivalent according to the Unicode standard are *not* equal unless, after replacement of each *UnicodeEscapeSequence*, they are represented by the exact same sequence of code points.

**Syntax**

```
IdentifierName  ::
  IdentifierStart
  IdentifierName  IdentifierPart

IdentifierStart  ::
```
The definitions of the nonterminal `UnicodeEscapeSequence` is given in 11.8.4.

NOTE 2 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”.

11.6.1 Identifier Names

11.6.1.1 Static Semantics: Early Errors

`IdentifierStart :: \ UnicodeEscapeSequence`

- It is a Syntax Error if `SV(UnicodeEscapeSequence)` is none of "$", or "_", or the UTF16Encoding of a code point matched by the `UnicodeIDStart` lexical grammar production.

`IdentifierPart :: \ UnicodeEscapeSequence`

- It is a Syntax Error if `SV(UnicodeEscapeSequence)` is none of "$", or "_", or the UTF16Encoding of either <ZWNJ> or <ZWJ>, or the UTF16Encoding of a Unicode code point that would be matched by the `UnicodeIDContinue` lexical grammar production.

11.6.1.2 Static Semantics: StringValue

`IdentifierName ::`
1. Return the String value consisting of the sequence of code units corresponding to IdentifierName. In determining the sequence any occurrences of \ UnicodeEscapeSequence are first replaced with the code point represented by the UnicodeEscapeSequence and then the code points of the entire IdentifierName are converted to code units by UTF16Encoding each code point.

### 11.6.2 Reserved Words

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

**Syntax**

```
ReservedWord ::
    Keyword
    FutureReservedWord
    NullLiteral
    BooleanLiteral
```

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

### 11.6.2.1 Keywords

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

**Syntax**

```
Keyword ::
    await
    break
    case
catch
class
case
class
class
const
continue
debugger
default
deleletedo
do
definedelse
exportextends
finally
for
for
function
if
import
in
instanceof
new
return
super
switch
```
this throw try typeof
var void
while with
yield

NOTE In some contexts yield and await are given the semantics of an Identifier. See 12.1.1. In strict mode code, let and static are treated as reserved words through static semantic restrictions (see 12.1.1, 13.3.1.1, 13.7.5.1, and 14.5.1) rather than the lexical grammar.

11.6.2.2 Future Reserved Words

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

Syntax

FutureReservedWord ::
   enum

NOTE Use of the following tokens within strict mode code is also reserved. That usage is restricted using static semantic restrictions (see 12.1.1) rather than the lexical grammar:

   implements package protected
   interface private public

11.7 Punctuators

Syntax

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

11.8.1 Null Literals

Syntax

NullLiteral ::
   null

11.8.2 Boolean Literals

Syntax

BooleanLiteral ::
   true
   false

11.8.3 Numeric Literals

Syntax

NumericLiteral ::
   DecimalLiteral
   BinaryIntegerLiteral
   OctalIntegerLiteral
   HexIntegerLiteral

DecimalLiteral ::
   DecimalIntegerLiteral . DecimalDigits<opt> ExponentPart<opt>
DecimalDigits ExponentPart\textsubscript{opt}
DecimalIntegerLiteral ExponentPart\textsubscript{opt}

DecimalIntegerLiteral ::
  0
  NonZeroDigit DecimalDigits\textsubscript{opt}

DecimalDigits ::
  DecimalDigit
  DecimalDigits DecimalDigit

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

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

ExponentPart ::
  ExponentIndicator SignedInteger

ExponentIndicator :: one of
  e E

SignedInteger ::
  DecimalDigits
  + DecimalDigits
  - DecimalDigits

BinaryIntegerLiteral ::
  0b BinaryDigits
  0B BinaryDigits

BinaryDigits ::
  BinaryDigit
  BinaryDigits BinaryDigit

BinaryDigit :: one of
  0 1

OctalIntegerLiteral ::
  0o OctalDigits
  0O OctalDigits

OctalDigits ::
OctalDigit
OctalDigits OctalDigit

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

HexIntegerLiteral ::
  0x  HexDigits
  0X  HexDigits

HexDigits ::
  HexDigit
  HexDigits  HexDigit

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

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

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

A conforming implementation, when processing strict mode code, 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.

11.8.3.1 Static Semantics: MV

A numeric literal stands for a value of the Number type. This value is determined in two steps: first, a mathematical value (MV) is derived from the literal; second, this mathematical value is rounded as described below.

- The MV of NumericLiteral :: DecimalLiteral is the MV of DecimalLiteral.
- The MV of NumericLiteral :: BinaryIntegerLiteral is the MV of BinaryIntegerLiteral.
- The MV of NumericLiteral :: OctalIntegerLiteral is the MV of OctalIntegerLiteral.
- The MV of NumericLiteral :: HexIntegerLiteral is the MV of HexIntegerLiteral.
- The MV of DecimalLiteral :: DecimalIntegerLiteral is the MV of DecimalIntegerLiteral.
- The MV of DecimalLiteral :: DecimalIntegerLiteral DecimalDigits is the MV of DecimalIntegerLiteral plus (the MV of DecimalDigits \( \times 10^n \)), where \( n \) is the number of code points in DecimalDigits.
- The MV of DecimalLiteral :: DecimalIntegerLiteral ExponentPart is the MV of \( \text{DecimalIntegerLiteral} \times 10^e \), where \( e \) is the MV of ExponentPart.
• The MV of \texttt{DecimalLiteral} :: \texttt{DecimalIntegerLiteral} . \texttt{DecimalDigits} \texttt{ExponentPart} is (the MV of \texttt{DecimalIntegerLiteral} plus (the MV of \texttt{DecimalDigits} \times 10^{-n})) \times 10^{e}, where \( n \) is the number of code points in \texttt{DecimalDigits} and \( e \) is the MV of \texttt{ExponentPart}.
• The MV of \texttt{DecimalLiteral} :: . \texttt{DecimalDigits} is the MV of \texttt{DecimalDigits} \times 10^{-n}, where \( n \) is the number of code points in \texttt{DecimalDigits}.
• The MV of \texttt{DecimalLiteral} :: . \texttt{DecimalDigits} \texttt{ExponentPart} is the MV of \texttt{DecimalDigits} \times 10^{e-n}, where \( n \) is the number of code points in \texttt{DecimalDigits} and \( e \) is the MV of \texttt{ExponentPart}.
• The MV of \texttt{DecimalLiteral} :: \texttt{DecimalIntegerLiteral} is the MV of \texttt{DecimalIntegerLiteral}.
• The MV of \texttt{DecimalLiteral} :: \texttt{DecimalIntegerLiteral} \texttt{ExponentPart} is the MV of \texttt{DecimalIntegerLiteral} \times 10^{e}, where \( e \) is the MV of \texttt{ExponentPart}.
• The MV of \texttt{DecimalIntegerLiteral} :: \texttt{0} is 0.
• The MV of \texttt{DecimalIntegerLiteral} :: \texttt{NonZeroDigit} is the MV of \texttt{NonZeroDigit}.
• The MV of \texttt{DecimalIntegerLiteral} :: \texttt{NonZeroDigit} \texttt{DecimalDigits} is (the MV of \texttt{NonZeroDigit} \times 10^{n}) plus the MV of \texttt{DecimalDigits}, where \( n \) is the number of code points in \texttt{DecimalDigits}.
• The MV of \texttt{DecimalDigits} :: \texttt{DecimalDigit} is the MV of \texttt{DecimalDigit}.
• The MV of \texttt{DecimalDigits} :: \texttt{DecimalDigits} \texttt{DecimalDigit} is (the MV of \texttt{DecimalDigits} \times 10) plus the MV of \texttt{DecimalDigit}.
• The MV of \texttt{ExponentPart} :: \texttt{ExponentIndicator} \texttt{SignedInteger} is the MV of \texttt{SignedInteger}.
• The MV of \texttt{SignedInteger} :: \texttt{DecimalDigits} is the MV of \texttt{DecimalDigits}.
• The MV of \texttt{SignedInteger} :: + \texttt{DecimalDigits} is the MV of \texttt{DecimalDigits}.
• The MV of \texttt{SignedInteger} :: - \texttt{DecimalDigits} is the negative of the MV of \texttt{DecimalDigits}.
• The MV of \texttt{DecimalDigit} :: \texttt{0} or of \texttt{HexDigit} :: \texttt{0} or of \texttt{OctalDigit} :: \texttt{0} or of \texttt{BinaryDigit} :: \texttt{0} is 0.
• The MV of \texttt{DecimalDigit} :: \texttt{1} or of \texttt{NonZeroDigit} :: \texttt{1} or of \texttt{HexDigit} :: \texttt{1} or of \texttt{OctalDigit} :: \texttt{1} or of \texttt{BinaryDigit} :: \texttt{1} is 1.
• The MV of \texttt{DecimalDigit} :: \texttt{2} or of \texttt{NonZeroDigit} :: \texttt{2} or of \texttt{HexDigit} :: \texttt{2} or of \texttt{OctalDigit} :: \texttt{2} is 2.
• The MV of \texttt{DecimalDigit} :: \texttt{3} or of \texttt{NonZeroDigit} :: \texttt{3} or of \texttt{HexDigit} :: \texttt{3} or of \texttt{OctalDigit} :: \texttt{3} is 3.
• The MV of \texttt{DecimalDigit} :: \texttt{4} or of \texttt{NonZeroDigit} :: \texttt{4} or of \texttt{HexDigit} :: \texttt{4} or of \texttt{OctalDigit} :: \texttt{4} is 4.
• The MV of \texttt{DecimalDigit} :: \texttt{5} or of \texttt{NonZeroDigit} :: \texttt{5} or of \texttt{HexDigit} :: \texttt{5} or of \texttt{OctalDigit} :: \texttt{5} is 5.
• The MV of \texttt{DecimalDigit} :: \texttt{6} or of \texttt{NonZeroDigit} :: \texttt{6} or of \texttt{HexDigit} :: \texttt{6} or of \texttt{OctalDigit} :: \texttt{6} is 6.
• The MV of \texttt{DecimalDigit} :: \texttt{7} or of \texttt{NonZeroDigit} :: \texttt{7} or of \texttt{HexDigit} :: \texttt{7} or of \texttt{OctalDigit} :: \texttt{7} is 7.
• The MV of \texttt{DecimalDigit} :: \texttt{8} or of \texttt{NonZeroDigit} :: \texttt{8} or of \texttt{HexDigit} :: \texttt{8} is 8.
• The MV of \texttt{DecimalDigit} :: \texttt{9} or of \texttt{NonZeroDigit} :: \texttt{9} or of \texttt{HexDigit} :: \texttt{9} is 9.
• The MV of \texttt{HexDigit} :: \texttt{a} or of \texttt{HexDigit} :: \texttt{A} is 10.
• The MV of \texttt{HexDigit} :: \texttt{b} or of \texttt{HexDigit} :: \texttt{B} is 11.
• The MV of \texttt{HexDigit} :: \texttt{c} or of \texttt{HexDigit} :: \texttt{C} is 12.
• The MV of \texttt{HexDigit} :: \texttt{d} or of \texttt{HexDigit} :: \texttt{D} is 13.
• The MV of \texttt{HexDigit} :: \texttt{e} or of \texttt{HexDigit} :: \texttt{E} is 14.
• The MV of \texttt{HexDigit} :: \texttt{f} or of \texttt{HexDigit} :: \texttt{F} is 15.
• The MV of \texttt{BinaryIntegerLiteral} :: \texttt{0b} \texttt{BinaryDigits} is the MV of \texttt{BinaryDigits}.
• The MV of \texttt{BinaryIntegerLiteral} :: \texttt{0B} \texttt{BinaryDigits} is the MV of \texttt{BinaryDigits}.
• The MV of BinaryDigits :: BinaryDigit is the MV of BinaryDigit.
• The MV of BinaryDigits :: BinaryDigits BinaryDigit is (the MV of BinaryDigits × 2) plus the MV of BinaryDigit.
• The MV of OctalIntegerLiteral :: 0o OctalDigits is the MV of OctalDigits.
• The MV of OctalIntegerLiteral :: 0O OctalDigits is the MV of OctalDigits.
• The MV of OctalDigits :: OctalDigit is the MV of OctalDigit.
• The MV of OctalDigits :: OctalDigits OctalDigit is (the MV of OctalDigits × 8) plus the MV of OctalDigit.
• The MV of HexIntegerLiteral :: 0x HexDigits is the MV of HexDigits.
• The MV of HexIntegerLiteral :: 0X HexDigits is the MV of HexDigits.
• The MV of HexDigits :: HexDigit is the MV of HexDigit.
• The MV of HexDigits :: HexDigits HexDigit is (the MV of HexDigits × 16) plus the MV of HexDigit.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the Number value for the MV (as specified in 6.1.6), unless the literal is a DecimalLiteral and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th significant digit position. A digit is significant if it is not part of an ExponentPart and

• it is not 0; or
• there is a nonzero digit to its left and there is a nonzero digit, not in the ExponentPart, to its right.

### 11.8.4 String Literals

**NOTE 1** A string literal is zero or more Unicode code points enclosed in single or double quotes. Unicode code points may also be represented by an escape sequence. All code points may appear literally in a string literal except for the closing quote code points, U+005C (REVERSE SOLIDUS), U+000D (CARRIAGE RETURN), U+2028 (LINE SEPARATOR), U+2029 (PARAGRAPH SEPARATOR), 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 10.1.1. 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 ::
    " DoubleStringCharactersopt "
```
SingleStringCharacters\textsubscript{opt}:

DoubleStringCharacters ::
  DoubleStringCharacter DoubleStringCharacters\textsubscript{opt}

SingleStringCharacters ::
  SingleStringCharacter SingleStringCharacters\textsubscript{opt}

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

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

LineContinuation ::
  \ LineTerminatorSequence

EscapeSequence ::
  CharacterEscapeSequence
  \ [lookahead \not\in DecimalDigit]
  HexEscapeSequence
  UnicodeEscapeSequence

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

CharacterEscapeSequence ::
  SingleEscapeCharacter
  NonEscapeCharacter

SingleEscapeCharacter :: one of
  ' " \ b f n r t v

NonEscapeCharacter ::
  SourceCharacter but not one of EscapeCharacter or LineTerminator

EscapeCharacter ::
  SingleEscapeCharacter
  DecimalDigit
  x
The definition of the nonterminal *HexDigit* is given in 11.8.3. *SourceCharacter* is defined in 10.1.

**NOTE 2**

A line terminator code point cannot appear in a string literal, except as part of a *LineContinuation* to produce the empty code points sequence. The proper way to cause a line terminator code point to be part of the String value of a string literal is to use an escape sequence such as `\n` or `\u000A`.

### 11.8.4.1 Static Semantics: Early Errors

*UnicodeEscapeSequence :: u{ HexDigits }

- It is a Syntax Error if the MV of *HexDigits* > 0x10FFFF.

### 11.8.4.2 Static Semantics: String Value

*StringLiteral ::

```
" DoubleStringCharacters_{opt} "
'
SingleStringCharacters_{opt}
```

1. Return the String value whose elements are the SV of this *StringLiteral*.

### 11.8.4.3 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 code unit 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 11.8.3.

- The SV of *StringLiteral :: " "* is the empty code unit sequence.
- The SV of *StringLiteral :: ' '* is the empty code unit sequence.
- The SV of *StringLiteral :: " DoubleStringCharacters*"* is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral :: ' SingleStringCharacters* '* is the SV of *SingleStringCharacters*.**
The SV of DoubleStringCharacters :: DoubleStringCharacter is a sequence of one or two code units that is the SV of DoubleStringCharacter.

The SV of DoubleStringCharacters :: DoubleStringCharacter DoubleStringCharacters is a sequence of one or two code units that is the SV of DoubleStringCharacter followed by all the code units in the SV of DoubleStringCharacters in order.

The SV of SingleStringCharacters :: SingleStringCharacter is a sequence of one or two code units that is the SV of SingleStringCharacter.

The SV of SingleStringCharacters :: SingleStringCharacter SingleStringCharacters is a sequence of one or two code units that is the SV of SingleStringCharacter followed by all the code units in the SV of SingleStringCharacters in order.

The SV of DoubleStringCharacter :: SourceCharacter but not one of " or \ or LineTerminator is the UTF16Encoding of the code point value of SourceCharacter.

The SV of DoubleStringCharacter :: \
EscapeSequence is the SV of the EscapeSequence.

The SV of DoubleStringCharacter :: LineContinuation is the empty code unit sequence.

The SV of SingleStringCharacter :: SourceCharacter but not one of ’ or \ or LineTerminator is the UTF16Encoding of the code point value of SourceCharacter.

The SV of SingleStringCharacter :: \
EscapeSequence is the SV of the EscapeSequence.

The SV of SingleStringCharacter :: LineContinuation is the empty code unit sequence.

The SV of EscapeSequence :: CharacterEscapeSequence is the SV of the CharacterEscapeSequence.

The SV of EscapeSequence :: \0 is the code unit value 0.

The SV of EscapeSequence :: HexEscapeSequence is the SV of the HexEscapeSequence.

The SV of EscapeSequence :: UnicodeEscapeSequence is the SV of the UnicodeEscapeSequence.

The SV of CharacterEscapeSequence :: SingleEscapeCharacter is the code unit whose value is determined by the SingleEscapeCharacter according to Table 34.

<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>

Table 34: String Single Character Escape Sequences
• The SV of `CharacterEscapeSequence` :: `NonEscapeCharacter` is the SV of the `NonEscapeCharacter`.

• The SV of `NonEscapeCharacter` :: `SourceCharacter` but not one of `EscapeCharacter` or `LineTerminator` is the `UTF16Encoding` of the code point value of `SourceCharacter`.

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

• The SV of `UnicodeEscapeSequence` :: `u` `Hex4Digits` is the SV of `Hex4Digits`.

• The SV of `Hex4Digits` :: `HexDigit` `HexDigit` `HexDigit` `HexDigit` is the code unit value that is (0x1000 times the MV of the first `HexDigit`) plus (0x100 times the MV of the second `HexDigit`) plus (0x10 times the MV of the third `HexDigit`) plus the MV of the fourth `HexDigit`.

• The SV of `UnicodeEscapeSequence` :: `u{` `HexDigits` `}` is the `UTF16Encoding` of the MV of `HexDigits`.

### 11.8.5 Regular Expression Literals

A regular expression literal is an input element that is converted to a RegExp object (see 21.2) 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 21.2.3).

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 (21.2.1).

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

#### Syntax

```plaintext
RegularExpressionLiteral ::
   / RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody ::
   RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars ::
   [empty]
   RegularExpressionChars RegularExpressionChar
```
### Regular Expression First Char
```
RegularExpressionFirstChar ::
   RegularExpressionNonTerminator but not one of * or \ or / or [
   RegularExpressionBackslashSequence
   RegularExpressionClass
```

### Regular Expression Char
```
RegularExpressionChar ::
   RegularExpressionNonTerminator but not one of \ or / or [
   RegularExpressionBackslashSequence
   RegularExpressionClass
```

### Regular Expression Backslash Sequence
```
RegularExpressionBackslashSequence ::
   \ RegularExpressionNonTerminator
```

### Regular Expression Non Terminator
```
RegularExpressionNonTerminator ::
   SourceCharacter but not LineTerminator
```

### Regular Expression Class
```
RegularExpressionClass ::
   [ RegularExpressionClassChars ]
```

#### Regular Expression Class Chars
```
[empty]
```

### Regular Expression Class Char
```
RegularExpressionClassChar ::
   RegularExpressionNonTerminator but not one of ] or \
   RegularExpressionBackslashSequence
```

### Regular Expression Flags
```
RegularExpressionFlags ::
   [empty]
```

#### Regular Expression Flags Identifier Part
```
```

---

**NOTE 2**

Regular expression literals may not be empty; instead of representing an empty regular expression literal, the code unit sequence `//` starts a single-line comment. To specify an empty regular expression, use: `/(?::)/`. 

### 11.8.5.1 Static Semantics: Early Errors

#### Regular Expression Flags
```
RegularExpressionFlags :: RegularExpressionFlags IdentifierPart
```

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

### 11.8.5.2 Static Semantics: Body Text

#### Regular Expression Literal
```
RegularExpressionLiteral :: / RegularExpressionBody / RegularExpressionFlags
```

1. Return the source text that was recognized as `RegularExpressionBody`.

### 11.8.5.3 Static Semantics: FlagText

`RegularExpressionLiteral ::= / RegularExpressionBody / RegularExpressionFlags`

1. Return the source text that was recognized as `RegularExpressionFlags`.

### 11.8.6 Template Literal Lexical Components

**Syntax**

```
Template  ::=  
  NoSubstitutionTemplate  
  TemplateHead  

NoSubstitutionTemplate  ::=  
  ` TemplateCharacters{opt} `  

TemplateHead  ::=  
  ` TemplateCharacters{opt} ${

TemplateSubstitutionTail  ::=  
  TemplateMiddle  
  TemplateTail  

TemplateMiddle  ::=  
  } TemplateCharacters{opt} ${

TemplateTail  ::=  
  } TemplateCharacters{opt} `  

TemplateCharacters  ::=  
  TemplateCharacter TemplateCharacters{opt}  

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

A conforming implementation must not use the extended definition of `EscapeSequence` described in B.1.2 when parsing a `TemplateCharacter`. 
11.8.6.1 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 code unit values (SV, 11.8.4) contributed by the various parts of the template literal component. As part of this process, some Unicode code points within the template component are interpreted as having a mathematical value (MV, 11.8.3). In determining a TV, escape sequences are replaced by the UTF-16 code unit(s) of the Unicode code point represented by the escape sequence. The Template Raw Value (TRV) is similar to a Template Value with the difference that in TRVs escape sequences are interpreted literally.

- The TV and TRV of NoSubstitutionTemplate :: `` is the empty code unit sequence.
- The TV and TRV of TemplateHead :: ` ${} is the empty code unit sequence.
- The TV and TRV of TemplateMiddle :: } ${} is the empty code unit sequence.
- The TV and TRV of TemplateTail :: ` is the empty code unit sequence.
- The TV of NoSubstitutionTemplate :: ` TemplateCharacters ` is the TV of TemplateCharacters.
- The TV of TemplateHead :: ` TemplateCharacters ${} is the TV of TemplateCharacters.
- The TV of TemplateMiddle :: } TemplateCharacters ${} is the TV of TemplateCharacters.
- The TV of TemplateTail :: } TemplateCharacters ` is the TV of TemplateCharacters.
- The TV of TemplateCharacters :: TemplateCharacter TemplateCharacters is a sequence consisting of the code units in the TV of TemplateCharacter followed by all the code units in the TV of TemplateCharacters in order.
- The TV of TemplateCharacter :: SourceCharacter but not one of ` or \ or $ or LineTerminator is the UTF16Encoding of the code point value of SourceCharacter.
- The TV of TemplateCharacter :: $ is the code unit value 0x0024.
- The TV of TemplateCharacter :: \ EscapeSequence is the SV of EscapeSequence.
- The TV of TemplateCharacter :: LineContinuation is the TV of LineContinuation.
- The TV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- The TV of LineContinuation :: \ LineTerminatorSequence is the empty code unit sequence.
- The TRV of NoSubstitutionTemplate :: ` TemplateCharacters ` is the TRV of TemplateCharacters.
- The TRV of TemplateHead :: ` TemplateCharacters ${} is the TRV of TemplateCharacters.
- The TRV of TemplateMiddle :: } TemplateCharacters ${} is the TRV of TemplateCharacters.
- The TRV of TemplateTail :: } TemplateCharacters ` is the TRV of TemplateCharacters.
- The TRV of TemplateCharacters :: TemplateCharacter TemplateCharacters is a sequence consisting of the code units in the TRV of TemplateCharacter followed by all the code units in the TRV of TemplateCharacters, in order.
- The TRV of TemplateCharacter :: SourceCharacter but not one of ` or \ or $ or LineTerminator is the UTF16Encoding of the code point value of SourceCharacter.
• The TRV of TemplateCharacter :: $ is the code unit value 0x0024.
• The TRV of TemplateCharacter :: \ EscapeSequence is the sequence consisting of the code unit value 0x005C followed by the code units of TRV of EscapeSequence.
• The TRV of TemplateCharacter :: LineContinuation is the TRV of LineContinuation.
• The TRV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
• The TRV of EscapeSequence :: CharacterEscapeSequence is the TRV of the CharacterEscapeSequence.
• The TRV of EscapeSequence :: $\r$ is the code unit value 0x0030 (DIGIT ZERO).
• The TRV of EscapeSequence :: HexEscapeSequence is the TRV of the HexEscapeSequence.
• The TRV of EscapeSequence :: UnicodeEscapeSequence is the TRV of the UnicodeEscapeSequence.
• The TRV of CharacterEscapeSequence :: SingleEscapeCharacter is the TRV of the SingleEscapeCharacter.
• The TRV of CharacterEscapeSequence :: NonEscapeCharacter is the SV of the NonEscapeCharacter.
• The TRV of SingleEscapeCharacter :: : one of : " $ \b \f \n \r \t \v \" \b \f \n \r \t \v is the SV of the SourceCharacter that is that single code point.

• The TRV of HexEscapeSequence :: 0 HexDigit HexDigit is the sequence consisting of code unit value 0x0078 followed by TRV of the first HexDigit followed by the TRV of the second HexDigit.
• The TRV of UnicodeEscapeSequence :: u Hex4Digits is the sequence consisting of code unit value 0x0075 followed by TRV of Hex4Digits.
• The TRV of UnicodeEscapeSequence :: u{ HexDigits } is the sequence consisting of code unit value 0x0075 followed by code unit value 0x007B followed by TRV of HexDigits followed by code unit value 0x007D.
• The TRV of Hex4Digits :: HexDigit HexDigit HexDigit HexDigit is the sequence consisting of the TRV of the first HexDigit followed by the TRV of the second HexDigit followed by the TRV of the third HexDigit followed by the TRV of the fourth HexDigit.
• The TRV of HexDigits :: HexDigit is the TRV of HexDigit.
• The TRV of HexDigits :: HexDigits HexDigit is the sequence consisting of TRV of HexDigits followed by TRV of HexDigit.

• The TRV of a HexDigit is the SV of the SourceCharacter that is that HexDigit.
• The TRV of LineContinuation :: \ LineTerminatorSequence is the sequence consisting of the code unit value 0x005C followed by the code units of TRV of LineTerminatorSequence.
• The TRV of LineTerminatorSequence :: <LF> is the code unit value 0x000A.
• The TRV of LineTerminatorSequence :: <CR> is the code unit value 0x000A.
• The TRV of LineTerminatorSequence :: <LS> is the code unit value 0x2028.
• The TRV of LineTerminatorSequence :: <PS> is the code unit value 0x2029.
• The TRV of LineTerminatorSequence :: <CR><LF> is the sequence consisting of the code unit value 0x000A.

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

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.

11.9.1 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 11.

There are three basic rules of semicolon insertion:

1. When, as the source text is parsed from left to right, a token (called the offending token) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
   - The offending token is separated from the previous token by at least one LineTerminator.
   - The offending token is ).
   - The previous token is ) and the inserted semicolon would then be parsed as the terminating semicolon of a do-while statement (13.7.2).
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 13.7.4).

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

\[
\begin{align*}
\text{UpdateExpression} & : \text{Yield, Await} \\
\text{LeftHandSideExpression} & : [?\text{Yield, Await}] \quad [\text{no LineTerminator here}] ++ \\
\text{LeftHandSideExpression} & : [?\text{Yield, Await}] \quad [\text{no LineTerminator here}] --
\end{align*}
\]
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.

11.9.2 Examples of Automatic Semicolon Insertion

The source

```javascript
{ 1 2 } 3
```

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

```plaintext
a = b
++c
```

is transformed by automatic semicolon insertion into the following:

```plaintext
a = b;
++c;
```

| NOTE 2 | The token `++` is not treated as a postfix operator applying to the variable `b`, because a `LineTerminator` occurs between `b` and `++`. |

The source

```plaintext
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

```plaintext
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:

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

In the circumstance that an assignment statement must begin with a left parenthesis, it is a good idea for the programmer to provide an explicit semicolon at the end of the preceding statement rather than to rely on automatic semicolon insertion.

## 12 ECMAScript Language: Expressions

### 12.1 Identifiers

Syntax
IdentifierReference \([\text{Yield, Await}]\) :
  Identifier
  \([-\text{Yield}] \text{ yield}\)
  \([-\text{Await}] \text{ await}\)

BindingIdentifier \([\text{Yield, Await}]\) :
  Identifier
  \([-\text{Yield}] \text{ yield}\)
  \([-\text{Await}] \text{ await}\)

LabelIdentifier \([\text{Yield, Await}]\) :
  Identifier
  \([-\text{Yield}] \text{ yield}\)
  \([-\text{Await}] \text{ await}\)

Identifier :
  IdentifierName but not ReservedWord

### 12.1.1 Static Semantics: Early Errors

**BindingIdentifier** : **Identifier**

- It is a Syntax Error if the code matched by this production is contained in strict mode code and the String Value of **Identifier** is "arguments" or "eval".

**IdentifierReference** : **yield**

**BindingIdentifier** : **yield**

**LabelIdentifier** : **yield**

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

**IdentifierReference** : **await**

**BindingIdentifier** : **await**

**LabelIdentifier** : **await**

- It is a Syntax Error if the goal symbol of the syntactic grammar is **Module**.

**IdentifierReference** : **Identifier**

**BindingIdentifier** : **Identifier**

**LabelIdentifier** : **Identifier**

- It is a Syntax Error if this production has a \([\text{Yield}]\) Parameter and String Value of **Identifier** is "yield".

- It is a Syntax Error if this production has an \([\text{Await}]\) parameter and String Value of **Identifier** is "await".
Identifer : IdentifierName but not ReservedWord

- It is a Syntax Error if this phrase is contained in strict mode code and the StringValue of IdentifierName is: "implements", "interface", "let", "package", "private", "protected", "public", "static", or "yield".
- It is a Syntax Error if StringValue of IdentifierName is the same String value as the StringValue of any ReservedWord except for yield.

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

12.1.2 Static Semantics: BoundNames

BindingIdentifier : Identifier

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

BindingIdentifier : yield

1. Return a new List containing "yield".

BindingIdentifier : await

1. Return a new List containing "await".

12.1.3 Static Semantics: IsValidSimpleAssignmentTarget

IdentifierReference : Identifier

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

IdentifierReference : yield

1. Return true.

IdentifierReference : await

1. Return true.

12.1.4 Static Semantics: StringValue

IdentifierReference : yield

BindingIdentifier : yield
LabelIdentifier : \textit{yield}

1. Return "\textit{yield}".

IdentifierReference : \textit{await}
BindingIdentifier : \textit{await}
LabelIdentifier : \textit{await}

1. Return "\textit{await}".

Identifier : IdentifierName but not ReservedWord

1. Return the StringValue of IdentifierName.

\subsection*{12.1.5 Runtime Semantics: BindingInitialization}

With parameters \textit{value} and \textit{environment}.

\begin{quote}
\textbf{NOTE} \texttt{undefined} is passed for \textit{environment} to indicate that a \texttt{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 \ref{sec:9.2.12}). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.
\end{quote}

BindingIdentifier : Identifier

1. Let \textit{name} be StringValue of Identifier.
2. Return ? \texttt{InitializeBoundName}(\textit{name}, \textit{value}, \textit{environment}).

BindingIdentifier : \textit{yield}

1. Return ? \texttt{InitializeBoundName}(\textit{"yield"}, \textit{value}, \textit{environment}).

BindingIdentifier : \textit{await}

1. Return ? \texttt{InitializeBoundName}(\textit{"await"}, \textit{value}, \textit{environment}).

\subsubsection*{12.1.5.1 Runtime Semantics: \texttt{InitializeBoundName} ( \textit{name}, \textit{value}, \textit{environment} )}

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

### 12.1.6 Runtime Semantics: Evaluation

#### IdentifierReference : Identifier

1. Return ? \( \text{ResolveBinding} \) (StringValue of \( \text{Identifier} \)).

#### IdentifierReference : yield

1. Return ? \( \text{ResolveBinding}("yield") \).

#### IdentifierReference : await

1. Return ? \( \text{ResolveBinding}("await") \).

---

**NOTE 1**  
The result of evaluating an \( \text{IdentifierReference} \) is always a value of type \( \text{Reference} \).

**NOTE 2**  
In non-strict code, the keyword \( \text{yield} \) may be used as an identifier. Evaluating the \( \text{IdentifierReference} \) resolves the binding of \( \text{yield} \) as if it was an \( \text{Identifier} \). Early Error restriction ensures that such an evaluation only can occur for non-strict code.

### 12.2 Primary Expression

#### Syntax

\[
\text{PrimaryExpression} \[\text{Yield, Await}] : \\
\quad \text{this} \\
\quad \text{IdentifierReference} \[?\text{Yield, ?Await}] \\
\quad \text{Literal} \\
\quad \text{ArrayLiteral} \[?\text{Yield, ?Await}] \\
\quad \text{ObjectLiteral} \[?\text{Yield, ?Await}] \\
\quad \text{FunctionExpression} \\
\quad \text{ClassExpression} \[?\text{Yield, ?Await}] \\
\quad \text{GeneratorExpression} \\
\quad \text{AsyncFunctionExpression} \\
\quad \text{RegularExpressionLiteral} \\
\quad \text{TemplateLiteral} \[?\text{Yield, ?Await}] \\
\quad \text{CoverParenthesizedExpressionAndArrowParameterList} \[?\text{Yield, ?Await}] \\
\]
Supplemental Syntax

When processing an instance of the production

\[ \text{PrimaryExpression} : \text{CoverParenthesizedExpressionAndArrowParameterList} \]

the interpretation of \( \text{CoverParenthesizedExpressionAndArrowParameterList} \) is refined using the following grammar:

\[
\text{ParenthesizedExpression}[\text{Yield, Await}] : \\
( \text{Expression} [+\text{In}, \text{Yield}, \text{Await}] )
\]

12.2.1 Semantics

12.2.1.1 Static Semantics: CoveredParenthesizedExpression

\( \text{CoverParenthesizedExpressionAndArrowParameterList} : ( \text{Expression} ) \)

1. Return the result of parsing the lexical token stream matched by

\( \text{CoverParenthesizedExpressionAndArrowParameterList} \) using \( \text{ParenthesizedExpression} \) as the goal symbol with its \([\text{Yield}]\) and \([\text{Await}]\) parameters set to the values used when parsing this \( \text{CoverParenthesizedExpressionAndArrowParameterList} \).

12.2.1.2 Static Semantics: HasName

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

1. Let \( \text{expr} \) be CoveredParenthesizedExpression of

\( \text{CoverParenthesizedExpressionAndArrowParameterList} \).

2. If IsFunctionDefinition of \( \text{expr} \) is \textbf{false}, return \textbf{false}.

3. Return HasName of \( \text{expr} \).

12.2.1.3 Static Semantics: IsFunctionDefinition

\( \text{PrimaryExpression} : \\
\text{this} \)
1. Return `false`.

`PrimaryExpression : IdentifierReference`

1. Return `true`.

`PrimaryExpression : IdentifierReference`

1. Return `false`.

### 12.2.1.4 Static Semantics: IsIdentifierRef

`PrimaryExpression : IdentifierReference`

1. Return `true`.

`PrimaryExpression :

this

Literal
ArrayLiteral
ObjectLiteral
FunctionExpression
ClassExpression
GeneratorExpression
AsyncFunctionExpression
RegularExpressionLiteral
TemplateLiteral
CoverParenthesizedExpressionAndArrowParameterList`

1. Return `false`.

### 12.2.1.5 Static Semantics: IsValidSimpleAssignmentTarget

`PrimaryExpression :

this

Literal
ArrayLiteral
ObjectLiteral`
1. Return `false`.

**PrimaryExpression** : `CoverParenthesizedExpressionAndArrowParameterList`

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

### 12.2.2 The this Keyword

#### 12.2.2.1 Runtime Semantics: Evaluation

**PrimaryExpression** : `this`

1. Return `? ResolveThisBinding()`.

### 12.2.3 Identifier Reference

See 12.1 for `IdentifierReference`.

### 12.2.4 Literals

**Syntax**

**Literal** :

`NullLiteral`

`BooleanLiteral`

`NumericLiteral`

`StringLiteral`

#### 12.2.4.1 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 number whose value is MV of NumericLiteral as defined in 11.8.3.

Literal : StringLiteral

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

12.2.5 Array Initializer

NOTE An ArrayLiteral is an expression describing the initialization of an Array object, using a list, of zero or more expressions each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initializer is evaluated.

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an AssignmentExpression (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.

Syntax

ArrayLiteral[Yield, Await] :
[ Elisionopt ]
[ ElementList[?Yield, ?Await], Elisionopt ]

ElementList[Yield, Await] :
Elisionopt AssignmentExpression[+In, ?Yield, ?Await]
Elisionopt SpreadElement[?Yield, ?Await]
ElementList[?Yield, ?Await], Elisionopt AssignmentExpression[+In, ?Yield, ?Await]
ElementList[?Yield, ?Await], Elisionopt SpreadElement[?Yield, ?Await]

Elision :
,

Elision,
12.2.5.1 Static Semantics: ElisionWidth

\( Elision : \) ,

1. Return the numeric value 1.

\( Elision : Elision \),

1. Let \( \text{preceding} \) be the ElisionWidth of \( Elision \).
2. Return \( \text{preceding} + 1 \).

12.2.5.2 Runtime Semantics: ArrayAccumulation

With parameters \( array \) and \( nextIndex \).

\( ElementList : Elision \ AssignmentExpression \)

1. Let \( \text{padding} \) be the ElisionWidth of \( Elision \); if \( Elision \) is not present, use the numeric value zero.
2. Let \( \text{initResult} \) be the result of evaluating \( AssignmentExpression \).
3. Let \( \text{initValue} \) be \(? GetValue(\text{initResult})\).
4. Let \( \text{created} \) be \( \text{CreateDataProperty}(array, \text{ToString}(\text{ToUint32}(nextIndex + \text{padding})), \text{initValue}) \).
5. Assert: \( \text{created} \) is \text{true}.
6. Return \( nextIndex + \text{padding} + 1 \).

\( ElementList : Elision \ SpreadElement \)

1. Let \( \text{padding} \) be the ElisionWidth of \( Elision \); if \( Elision \) is not present, use the numeric value zero.
2. Return the result of performing ArrayAccumulation for \( SpreadElement \) with arguments \( array \) and \( nextIndex + \text{padding} \).

\( ElementList : ElementList , Elision \ AssignmentExpression \)

1. Let \( \text{postIndex} \) be the result of performing ArrayAccumulation for \( ElementList \) with arguments \( array \) and \( nextIndex \).
2. ReturnIfAbrupt(\( \text{postIndex} \)).
3. Let \( \text{padding} \) be the ElisionWidth of \( Elision \); if \( Elision \) is not present, use the numeric value zero.
4. Let \( \text{initResult} \) be the result of evaluating \( AssignmentExpression \).
5. Let \( \text{initValue} \) be \(? GetValue(\text{initResult})\).
6. Let \( \text{created} \) be \( \text{CreateDataProperty}(array, \text{ToString}(\text{ToUint32}(\text{postIndex} + \text{padding})), \text{initValue}) \).
7. Assert: \( \text{created} \) is \text{true}.
8. Return \( \text{postIndex} + \text{padding} + 1 \).
ElementList : ElementList , Elision SpreadElement

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

SpreadElement : ... AssignmentExpression

1. Let spreadRef be the result of evaluating AssignmentExpression.
2. Let spreadObj be ? GetValue(spreadRef).
3. Let iterator be ? GetIterator(spreadObj).
4. Repeat,
   a. Let next be ? IteratorStep(iterator).
   b. If next is false, return nextIndex.
   c. Let nextValue be ? IteratorValue(next).
   d. Let status be CreateDataProperty(array, ToString(ToUint32(nextIndex)), nextValue).
   e. Assert: status is true.
   f. Let nextIndex be nextIndex + 1.

NOTE CreateDataProperty is used to ensure that own properties are defined for the array even if the standard built-in Array prototype object has been modified in a manner that would preclude the creation of new own properties using [[Set]].

12.2.5.3 Runtime Semantics: Evaluation

ArrayLiteral : [ Elision ]

1. Let array be ! ArrayCreate(0).
2. Let pad be the ElisionWidth of Elision; if Elision is not present, use the numeric value zero.
3. Perform Set(array, "length", ToUint32(pad), false).
4. NOTE: The above Set cannot fail because of the nature of the object returned by ArrayCreate.
5. Return array.

ArrayLiteral : [ ElementList ]

1. Let array be ! ArrayCreate(0).
2. Let len be the result of performing ArrayAccumulation for ElementList with arguments array and 0.
3. ReturnIfAbrupt(len).
4. Perform Set(array, "length", ToUint32(len), false).
5. NOTE: The above Set cannot fail because of the nature of the object returned by ArrayCreate.
6. Return \( array \).

ArrayLiteral \( : [ \ ElementList , \ Elision \ ] \)

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

12.2.6 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

\[ ObjectLiteral[Yield, Await] : \]
\[
\begin{cases} 
\{ \} \\
\{ PropertyDefinitionList[Yield, Await] \} \\
\{ PropertyDefinitionList[Yield, Await], \}
\end{cases}
\]

\[ PropertyDefinitionList[Yield, Await] : \]

\[ PropertyDefinition[Yield, Await] \]

\[ PropertyDefinitionList[Yield, Await], PropertyDefinition[Yield, Await] \]

\[ PropertyDefinition[Yield, Await] : \]

\[ IdentifierReference[Yield, Await] \]

\[ CoverInitializedName[Yield, Await] \]

\[ PropertyName[Yield, Await] : AssignmentExpression[+In, Yield, Await] \]

\[ MethodDefinition[Yield, Await] \]

\[ PropertyName[Yield, Await] : \]

\[ LiteralPropertyName \]

\[ ComputedPropertyName[Yield, Await] \]

\[ LiteralPropertyName : \]

\[ IdentifierName \]
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 14.3.

NOTE 3  In certain contexts, ObjectLiteral is used as a cover grammar for a more restricted secondary grammar. The CoverInitializedName production is necessary to fully cover these secondary grammars. However, use of this production results in an early Syntax Error in normal contexts where an actual ObjectLiteral is expected.

12.2.6.1 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.

12.2.6.2 Static Semantics: ComputedPropertyContains

With parameter symbol.

PropertyName : LiteralPropertyName
1. Return false.

\[\text{PropertyName} : \text{ComputedPropertyName}\]

1. Return the result of \(\text{ComputedPropertyName} \text{ Contains } \text{symbol}\).

### 12.2.6.3 Static Semantics: Contains

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

\[\text{PropertyDefinition} : \text{MethodDefinition}\]

1. If \(\text{symbol}\) is \(\text{MethodDefinition}\), return true.
2. Return the result of \(\text{ComputedPropertyContains}\) for \(\text{MethodDefinition}\) with argument \(\text{symbol}\).

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

\[\text{LiteralPropertyName} : \text{IdentifierName}\]

1. If \(\text{symbol}\) is a \(\text{ReservedWord}\), return false.
2. If \(\text{symbol}\) is an \(\text{Identifier}\) and StringValue of \(\text{symbol}\) is the same value as the StringValue of \(\text{IdentifierName}\), return true.
3. Return false.

### 12.2.6.4 Static Semantics: IsComputedPropertyKey

\[\text{PropertyName} : \text{LiteralPropertyName}\]

1. Return false.

\[\text{PropertyName} : \text{ComputedPropertyName}\]

1. Return true.

### 12.2.6.5 Static Semantics: PropName

\[\text{PropertyDefinition} : \text{IdentifierReference}\]

1. Return StringValue of \(\text{IdentifierReference}\).

\[\text{PropertyDefinition} : \text{PropertyName} : \text{AssignmentExpression}\]

1. Return PropName of PropertyName.

\[\text{LiteralPropertyName} : \text{IdentifierName}\]
1. Return StringValue of IdentifierName.

LiteralPropertyName : StringLiteral

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

LiteralPropertyName : NumericLiteral

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

ComputedPropertyName : [ AssignmentExpression ]

1. Return empty.

12.2.6.6 Static Semantics: PropertyNameList

PropertyDefinitionList : PropertyDefinition

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

PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition

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

12.2.6.7 Runtime Semantics: Evaluation

ObjectLiteral : {  }

1. Return ObjectCreate(%ObjectPrototype%).

ObjectLiteral :

  { PropertyDefinitionList  }

  { PropertyDefinitionList ,  }

1. Let obj be ObjectCreate(%ObjectPrototype%).
2. Let status be the result of performing PropertyDefinitionEvaluation of PropertyDefinitionList with arguments obj and true.
3. ReturnIfAbrupt(status).
4. Return obj.

LiteralPropertyName : IdentifierName
1. Return StringValue of *IdentifierName*.

**LiteralPropertyName** : StringLiteral

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

**LiteralPropertyName** : NumericLiteral

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.

**ComputedPropertyName** : [ AssignmentExpression ]

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

### 12.2.6.8 Runtime Semantics: PropertyDefinitionEvaluation

With parameters *object* and *enumerable*.

**PropertyDefinitionList** : PropertyDefinitionList , PropertyDefinition

1. Let *status* be the result of performing PropertyDefinitionEvaluation of *PropertyDefinitionList* with arguments *object* and *enumerable*.
2. ReturnIfAbrupt(status).
3. Return the result of performing PropertyDefinitionEvaluation of *PropertyDefinition* with arguments *object* and *enumerable*.

**PropertyDefinition** : IdentifierReference

1. Let *propName* be StringValue of *IdentifierReference*.
2. Let *exprValue* be the result of evaluating *IdentifierReference*.
4. Assert: *enumerable* is true.
5. Return CreateDataPropertyOrThrow(*object*, *propName*, *propValue*).

**PropertyDefinition** : PropertyName : AssignmentExpression

1. Let *propKey* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propKey*).
3. Let *exprValueRef* be the result of evaluating AssignmentExpression.
4. Let *propValue* be ? *GetValue*(exprValueRef).
5. If *IsAnonymousFunctionDefinition*(AssignmentExpression) is true, then
a. Let `hasNameProperty` be `HasOwnProperty(propValue, "name")`.

b. If `hasNameProperty` is `false`, perform `SetFunctionName(propValue, propKey)`.

6. Assert: `enumerable` is `true`.
7. Return `CreateDataPropertyOrThrow(object, propKey, propValue)`.

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

### 12.2.7 Function Defining Expressions

See 14.1 for `PrimaryExpression : FunctionExpression`.

See 14.4 for `PrimaryExpression : GeneratorExpression`.

See 14.5 for `PrimaryExpression : ClassExpression`.

See 14.6 for `PrimaryExpression : AsyncFunctionExpression`.

### 12.2.8 Regular Expression Literals

**Syntax**

See 11.8.5.

#### 12.2.8.1 Static Semantics: Early Errors

`PrimaryExpression : RegularExpressionLiteral`

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

#### 12.2.8.2 Runtime Semantics: Evaluation

`PrimaryExpression : RegularExpressionLiteral`

1. Let `pattern` be the String value consisting of the `UTF16Encoding` of each code point of BodyText of `RegularExpressionLiteral`.
2. Let `flags` be the String value consisting of the `UTF16Encoding` of each code point of FlagText of `RegularExpressionLiteral`.
3. Return `RegExpCreate(pattern, flags)`.

### 12.2.9 Template Literals
12.2.9.1 Static Semantics: TemplateStrings

With parameter `raw`.

TemplateLiteral : NoSubstitutionTemplate

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

TemplateLiteral : TemplateHead Expression TemplateSpans

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

TemplateSpans : TemplateTail

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

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

TemplateMiddleList : TemplateMiddle  Expression

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

TemplateMiddleList : TemplateMiddleList  TemplateMiddle  Expression

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

12.2.9.2 Runtime Semantics: ArgumentListEvaluation
TemplateLiteral : NoSubstitutionTemplate

1. Let templateLiteral be this TemplateLiteral.
2. Let siteObj be GetTemplateObject(templateLiteral).
3. Return a List containing the one element which is siteObj.

TemplateLiteral : TemplateHead  Expression  TemplateSpans

1. Let templateLiteral be this TemplateLiteral.
2. Let siteObj be GetTemplateObject(templateLiteral).
3. Let firstSubRef be the result of evaluating Expression.
4. Let firstSub be ? GetValue(firstSubRef).
5. Let restSub be SubstitutionEvaluation of TemplateSpans.
6. Return IfAbrupt(restSub).
7. Assert: restSub is a List.
8. Return a List whose first element is siteObj, whose second elements is firstSub, and whose subsequent elements are the elements of restSub, in order. restSub may contain no elements.

### 12.2.9.3 Runtime Semantics: GetTemplateObject (templateLiteral)

The abstract operation GetTemplateObject is called with a Parse Node, templateLiteral, as an argument. It performs the following steps:

1. Let rawStrings be TemplateStrings of templateLiteral with argument true.
2. Let realm be the current Realm Record.
3. Let templateRegistry be realm.[[TemplateMap]].
4. For each element \( e \) of templateRegistry, do
   a. If \( e.[[[Strings]]] \) and rawStrings contain the same values in the same order, then
      i. Return \( e.[[[Array]]] \).
5. Let cookedStrings be TemplateStrings of templateLiteral with argument false.
6. Let count be the number of elements in the List cookedStrings.
7. Let template be ! ArrayCreate(count).
8. Let rawObj be ! ArrayCreate(count).
9. Let index be 0.
10. Repeat, while index < count
    a. Let prop be ! ToString(index).
    b. Let cookedValue be the String value cookedStrings[index].
    c. Call template.[[[DefineOwnProperty]]][prop, PropertyDescriptor{[[Value]]: cookedValue,
                   [[Writable]]: false, [[Enumerable]]: true, [[Configurable]]: false}).
    d. Let rawValue be the String value rawStrings[index].
    e. Call rawObj.[[[DefineOwnProperty]]][prop, PropertyDescriptor{[[Value]]: rawValue, [[Writable]]: false, [[Enumerable]]: true, [[Configurable]]: false}).
    f. Let index be index+1.
11. Perform SetIntegrityLevel(rawObj, "frozen").
12. Call template.[[[DefineOwnProperty]]]"raw", PropertyDescriptor{[[Value]]: rawObj, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}).
13. Perform SetIntegrityLevel(template, "frozen").
14. Append the Record{[[Strings]]: rawStrings, [[Array]]: template} to templateRegistry.
15. 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 (12.2.9.5). 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.

12.2.9.4 Runtime Semantics: SubstitutionEvaluation

TemplateSpans : TemplateTail

1. Return a new empty List.

TemplateSpans : TemplateMiddleList TemplateTail

1. Return the result of SubstitutionEvaluation of TemplateMiddleList.

TemplateMiddleList : TemplateMiddle Expression

1. Let subRef be the result of evaluating Expression.
2. Let sub be ? GetValue(subRef).
3. Return a List containing only sub.

TemplateMiddleList : TemplateMiddleList TemplateMiddle Expression

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

12.2.9.5 Runtime Semantics: Evaluation

TemplateLiteral : NoSubstitutionTemplate

1. Return the String value whose code units are the elements of the TV of NoSubstitutionTemplate as defined in 11.8.6.

TemplateLiteral : TemplateHead Expression TemplateSpans

1. Let head be the TV of TemplateHead as defined in 11.8.6.
2. Let sub be the result of evaluating Expression.
3. ReturnIfAbrupt(sub).
4. Let middle be ? ToString(sub).
5. Let tail be the result of evaluating TemplateSpans.
6. ReturnIfAbrupt(tail).
7. Return the String value whose code units are the elements of *head* followed by the elements of *middle* followed by the elements of *tail*.

**NOTE 1**
The string conversion semantics applied to the *Expression* value are like `String.prototype.concat` rather than the `+` operator.

`TemplateSpans : TemplateTail`

1. Let *tail* be the TV of *TemplateTail* as defined in 11.8.6.
2. Return the String consisting of the code units of *tail*.

`TemplateSpans : TemplateMiddleList TemplateTail`

1. Let *head* be the result of evaluating *TemplateMiddleList*.
2. ReturnIfAbrupt(*head*).
3. Let *tail* be the TV of *TemplateTail* as defined in 11.8.6.
4. Return the String value whose code units are the elements of *head* followed by the elements of *tail*.

`TemplateMiddleList : TemplateMiddle Expression`

1. Let *head* be the TV of *TemplateMiddle* as defined in 11.8.6.
2. Let *sub* be the result of evaluating *Expression*.
3. ReturnIf Abrupt(*sub*).
4. Let *middle* be ? `ToString(*sub*)`.
5. Return the sequence of code units consisting of the code units of *head* followed by the elements of *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 11.8.6.
4. Let *sub* be the result of evaluating *Expression*.
5. ReturnIfAbrupt(*sub*).
6. Let *last* be ? `ToString(*sub*)`.
7. Return the sequence of code units consisting of the elements of *rest* followed by the code units of *middle* followed by the elements of *last*.

**NOTE 3**
The string conversion semantics applied to the *Expression* value are like `String.prototype.concat` rather than the `+` operator.
12.2.10 The Grouping Operator

12.2.10.1 Static Semantics: Early Errors

\textit{PrimaryExpression} : \textit{CoverParenthesizedExpressionAndArrowParameterList}

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

12.2.10.2 Static Semantics: IsFunctionDefinition

\textit{ParenthesizedExpression} : ( \textit{Expression} )

1. Return IsFunctionDefinition of \textit{Expression}.

12.2.10.3 Static Semantics: IsValidSimpleAssignmentTarget

\textit{ParenthesizedExpression} : ( \textit{Expression} )

1. Return IsValidSimpleAssignmentTarget of \textit{Expression}.

12.2.10.4 Runtime Semantics: Evaluation

\textit{PrimaryExpression} : \textit{CoverParenthesizedExpressionAndArrowParameterList}

1. Let \textit{expr} be CoveredParenthesizedExpression of \textit{CoverParenthesizedExpressionAndArrowParameterList}. 
2. Return the result of evaluating \textit{expr}.

\textit{ParenthesizedExpression} : ( \textit{Expression} )

1. Return the result of evaluating \textit{Expression}. This may be of type \texttt{Reference}.

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

12.3 Left-Hand-Side Expressions

\textbf{Syntax}

\textit{MemberExpression}[\texttt{Yield, Await}] :
PrimaryExpression[?Yield, ?Await]
MemberExpression[?Yield, ?Await]  [ Expression[+In, ?Yield, ?Await] ]
MemberExpression[?Yield, ?Await]  . IdentifierName
MemberExpression[?Yield, ?Await]  TemplateLiteral[?Yield, ?Await]
SuperProperty[?Yield, ?Await]
MetaProperty

SuperProperty[Yield, Await]:
  super  [ Expression[+In, ?Yield, ?Await] ]
  super  . IdentifierName

MetaProperty:
  NewTarget

NewTarget:
  new  . target

NewExpression[Yield, Await]:
  MemberExpression[?Yield, ?Await]
  new  NewExpression[?Yield, ?Await]

CallExpression[Yield, Await]:
  CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await]
  SuperCall[?Yield, ?Await]
  CallExpression[?Yield, ?Await]  [ Expression[+In, ?Yield, ?Await] ]
  CallExpression[?Yield, ?Await]  . IdentifierName

SuperCall[Yield, Await]:
  super  Arguments[?Yield, ?Await]

Arguments[Yield, Await]:
  ( )
  ( ArgumentList[?Yield, ?Await] , )

ArgumentList[Yield, Await]:
  AssignmentExpression[+In, ?Yield, ?Await]
  ...  AssignmentExpression[+In, ?Yield, ?Await]
Supplemental Syntax

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

```
CallMemberExpression[Yield, Await] :
    MemberExpression[Yield, Await] Arguments[Yield, Await]
```

12.3.1 Static Semantics

12.3.1.1 Static Semantics: CoveredCallExpression

```
CallExpression : CoverCallExpressionAndAsyncArrowHead
```

1. Return the result of parsing the lexical token stream matched by `CoverCallExpressionAndAsyncArrowHead` using `CallMemberExpression` as the goal symbol with its [Yield] and [Await] parameters set to the values used when parsing `CoverCallExpressionAndAsyncArrowHead`.

12.3.1.2 Static Semantics: Contains

With parameter `symbol`.

```
MemberExpression : MemberExpression . IdentifierName
```

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

```
SuperProperty : super . IdentifierName
```

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

**CallExpression**: `CallExpression . IdentifierName`

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

### 12.3.1.3 Static Semantics: IsFunctionDefinition

**MemberExpression**:  
- `MemberExpression [ Expression ]`
- `MemberExpression . IdentifierName`
- `MemberExpression TemplateLiteral`
- `SuperProperty`
- `MetaProperty`
  - `new MemberExpression Arguments`

**NewExpression**:  
- `new NewExpression`

**LeftHandSideExpression**:  
- `CallExpression`

1. Return `false`.

### 12.3.1.4 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

1. Return false.

12.3.1.5 Static Semantics: IsIdentifierRef

MemberExpression:
   MemberExpression [ Expression ]
   MemberExpression . IdentifierName
   MemberExpression TemplateLiteral
   SuperProperty
   MetaProperty
   new MemberExpression Arguments

NewExpression:
   new NewExpression

LeftHandSideExpression:
   CallExpression

1. Return false.

12.3.1.6 Static Semantics: IsValidSimpleAssignmentTarget

CallExpression:
   CallExpression [ Expression ]
   CallExpression . IdentifierName

MemberExpression:
   MemberExpression [ Expression ]
   MemberExpression . IdentifierName
   SuperProperty

1. Return true.

CallExpression:
   CoverCallExpressionAndAsyncArrowHead
   SuperCall
   CallExpression Arguments
**CallExpression TemplateLiteral**

*NewExpression :*

    new NewExpression

*MemberExpression : *

    MemberExpression TemplateLiteral
    new MemberExpression Arguments

*NewTarget : *

    new . target

1. Return **false**.

### 12.3.2 Property Accessors

**NOTE** Properties are accessed by name, using either the dot notation:

*MemberExpression . IdentifierName*

*CallExpression . IdentifierName*

or the bracket notation:

*MemberExpression [ Expression ]*

*CallExpression [ Expression ]*

The dot notation is explained by the following syntactic conversion:

*MemberExpression . IdentifierName*

is identical in its behaviour to

*MemberExpression [ <identifier-name-string> ]*

and similarly

*CallExpression . IdentifierName*

is identical in its behaviour to

*CallExpression [ <identifier-name-string> ]*

where `<identifier-name-string>` is the result of evaluating `StringValue` of `IdentifierName`.

### 12.3.2.1 Runtime Semantics: Evaluation
**MemberExpression** : MemberExpression [ Expression ]

1. Let baseReference be the result of evaluating MemberExpression.
2. Let baseValue be ? GetValue(baseReference).
3. Let propertyNameReference be the result of evaluating Expression.
4. Let propertyNameValue be ? GetValue(propertyNameReference).
5. Let bv be ? RequireObjectCoercible(baseValue).
6. Let propertyKey be ? ToPropertyKey(propertyNameValue).
7. If the code matched by this MemberExpression is strict mode code, let strict be true, else let strict be false.
8. Return a value of type Reference whose base value component is bv, whose referenced name component is propertyKey, and whose strict reference flag is strict.

**MemberExpression** : MemberExpression . IdentifierName

1. Let baseReference be the result of evaluating MemberExpression.
2. Let baseValue be ? GetValue(baseReference).
4. Let propertyNameString be StringValue of IdentifierName.
5. If the code matched by this MemberExpression is strict mode code, let strict be true, else let strict be false.
6. Return a value of type Reference whose base value component is bv, whose referenced name component is propertyNameString, and whose strict reference flag is strict.

**CallExpression** : CallExpression [ Expression ]

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

**CallExpression** : CallExpression . IdentifierName

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

### 12.3.3 The new Operator

#### 12.3.3.1 Runtime Semantics: Evaluation

**NewExpression** : new NewExpression


**MemberExpression** : new MemberExpression Arguments
1. Return ? \texttt{EvaluateNew}(\textit{MemberExpression}, \textit{Arguments}).

12.3.3.1.1 Runtime Semantics: \texttt{EvaluateNew} (\textit{constructExpr}, \textit{arguments})

The abstract operation \texttt{EvaluateNew} with arguments \textit{constructExpr}, and \textit{arguments} performs the following steps:

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

12.3.4 Function Calls

12.3.4.1 Runtime Semantics: Evaluation

\textit{CallExpression} : \texttt{CoverCallExpressionAndAsyncArrowHead}

1. Let \textit{expr} be \texttt{CoveredCallExpression} of \texttt{CoverCallExpressionAndAsyncArrowHead}.
2. Let \textit{memberExpr} be the \textit{MemberExpression} of \textit{expr}.
3. Let \textit{arguments} be the \textit{Arguments} of \textit{expr}.
4. Let \textit{ref} be the result of evaluating \textit{memberExpr}.
5. Let \textit{func} be ? \texttt{GetValue}(\textit{ref}).
6. If \texttt{Type}(\textit{ref}) is \texttt{Reference} and \texttt{IsPropertyReference}(\textit{ref}) is \texttt{false} and \texttt{GetReferencedName}(\textit{ref}) is "eval", then
   a. If \texttt{SameValue}(\textit{func}, \texttt{%eval%}) is \texttt{true}, then
      i. Let \textit{argList} be ? \texttt{ArgumentListEvaluation}(\textit{arguments}).
      ii. If \textit{argList} has no elements, return \texttt{undefined}.
      iii. Let \textit{evalText} be the first element of \textit{argList}.
      iv. If the source code matching this \textit{CallExpression} is strict mode code, let \textit{strictCaller} be \texttt{true}.
         Otherwise let \textit{strictCaller} be \texttt{false}.
      v. Let \textit{evalRealm} be the current Realm Record.
      vi. Perform ? \texttt{HostEnsureCanCompileStrings}(\textit{evalRealm}, \textit{evalRealm}).
      vii. Return ? \texttt{PerformEval}(\textit{evalText}, \textit{evalRealm}, \textit{strictCaller}, \texttt{true}).
7. If \texttt{Type}(\textit{ref}) is \texttt{Reference}, then
   a. If \texttt{IsPropertyReference}(\textit{ref}) is \texttt{true}, then
i. Let \( \text{thisValue} \) be \( \text{GetThisValue}(\text{ref}) \).
b. Else the base of \( \text{ref} \) is an Environment Record,
   i. Let \( \text{refEnv} \) be \( \text{GetBase}(\text{ref}) \).
   ii. Let \( \text{thisValue} \) be \( \text{refEnv}.\text{WithBaseObject}() \).
8. Else \( \text{Type}(\text{ref}) \) is not Reference,
   a. Let \( \text{thisValue} \) be \text{undefined}.
9. Let \( \text{thisCall} \) be this CallExpression.
10. Let \( \text{tailCall} \) be \( \text{IsInTailPosition}(\text{thisCall}) \).
11. Return \( \text{EvaluateDirectCall}(\text{func}, \text{thisValue}, \text{arguments}, \text{tailCall}) \).

A CallExpression evaluation that executes step 6.a.vii is a direct eval.

\[
\text{CallExpression} : \text{CallExpression \ Arguments} \\
1. \text{Let } \text{ref} \text{ be the result of evaluating CallExpression.} \\
2. \text{Let } \text{thisCall} \text{ be this CallExpression.} \\
3. \text{Let } \text{tailCall} \text{ be IsInTailPosition(thisCall).} \\
4. \text{Return } \text{EvaluateCall}(\text{ref}, \text{Arguments}, \text{tailCall}).
\]

12.3.4.2 Runtime Semantics: EvaluateCall( \text{ref, arguments, tailPosition} )

The abstract operation EvaluateCall takes as arguments a value \( \text{ref} \), a Parse Node arguments, and a Boolean argument \( \text{tailPosition} \). It performs the following steps:

1. Let \( \text{func} \) be \( \text{? GetValue(ref)} \).
2. If \( \text{Type}(\text{ref}) \) is Reference, then
   a. If \( \text{IsPropertyReference}(\text{ref}) \) is true, then
      i. Let \( \text{thisValue} \) be \( \text{GetThisValue}(\text{ref}) \).
   b. Else the base of \( \text{ref} \) is an Environment Record,
      i. Let \( \text{refEnv} \) be \( \text{GetBase}(\text{ref}) \).
      ii. Let \( \text{thisValue} \) be \( \text{refEnv}.\text{WithBaseObject}() \).
3. Else \( \text{Type}(\text{ref}) \) is not Reference,
   a. Let \( \text{thisValue} \) be \text{undefined}.
4. Return ? EvaluateDirectCall(\text{func}, \text{thisValue}, \text{arguments}, \text{tailPosition}).

12.3.4.3 Runtime Semantics: EvaluateDirectCall( \text{func, thisValue, arguments, tailPosition} )

The abstract operation EvaluateDirectCall takes as arguments a value \( \text{func} \), a value \( \text{thisValue} \), a Parse Node arguments, and a Boolean argument \( \text{tailPosition} \). It performs the following steps:

1. Let \( \text{argList} \) be ? ArgumentListEvaluation(\text{arguments}).
2. If \( \text{Type(func)} \) is not Object, throw a \text{TypeError} exception.
3. If \texttt{IsCallable(func)} is \texttt{false}, throw a \texttt{TypeError} exception.
4. If \texttt{tailPosition} is \texttt{true}, perform \texttt{PrepareForTailCall}().
5. Let \texttt{result} be \texttt{Call(func, thisValue, argList)}.
6. Assert: If \texttt{tailPosition} is \texttt{true}, the above call will not return here, but instead evaluation will continue as if the following return has already occurred.
7. Assert: If \texttt{result} is not an abrupt completion, then \texttt{Type(result)} is an ECMAScript language type.
8. Return \texttt{result}.

12.3.5 The super Keyword

12.3.5.1 Runtime Semantics: Evaluation

\texttt{SuperProperty : super [ Expression ]}

1. Let \texttt{propertyNameReference} be the result of evaluating \texttt{Expression}.
2. Let \texttt{propertyNameValue} be \texttt{? GetValue(propertyNameReference)}.
3. Let \texttt{propertyKey} be \texttt{? ToPropertyKey(propertyNameValue)}.
4. If the code matched by this \texttt{SuperProperty} is strict mode code, let \texttt{strict} be \texttt{true}, else let \texttt{strict} be \texttt{false}.
5. Return \texttt{? MakeSuperPropertyReference(propertyKey, strict)}.

\texttt{SuperProperty : super . IdentifierName}

1. Let \texttt{propertyKey} be StringValue of \texttt{IdentifierName}.
2. If the code matched by this \texttt{SuperProperty} is strict mode code, let \texttt{strict} be \texttt{true}, else let \texttt{strict} be \texttt{false}.
3. Return \texttt{? MakeSuperPropertyReference(propertyKey, strict)}.

\texttt{SuperCall : super Arguments}

1. Let \texttt{newTarget} be \texttt{GetNewTarget}().
2. Assert: \texttt{Type(newTarget)} is Object.
3. Let \texttt{func} be \texttt{? GetSuperConstructor}().
4. Let \texttt{argList} be ArgumentListEvaluation of \texttt{Arguments}.
5. Return\texttt{IfAbrupt(argList)}.
6. Let \texttt{result} be \texttt{? Construct(func, argList, newTarget)}.
7. Let \texttt{thisER} be \texttt{GetThisEnvironment}().
8. Return \texttt{thisER}.BindThisValue(result).

12.3.5.2 Runtime Semantics: GetSuperConstructor()

The abstract operation \texttt{GetSuperConstructor} performs the following steps:

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

12.3.5.3 Runtime Semantics: MakeSuperPropertyReference (propertyKey, strict)

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

1. Let `env` be `GetThisEnvironment()`.
3. Let `actualThis` be `? env.GetThisBinding()`.
5. Let `bv` be `? RequireObjectCoercible(baseValue)`.
6. Return a value of type `Reference` that is a `Super Reference` whose base value component is `bv`, whose referenced name component is `propertyKey`, whose thisValue component is `actualThis`, and whose strict reference flag is `strict`.

12.3.6 Argument Lists

NOTE The evaluation of an argument list produces a `List` of values.

12.3.6.1 Runtime Semantics: ArgumentListEvaluation
Arguments : ( )

1. Return a new empty `List`.

ArgumentList : AssignmentExpression

1. Let `ref` be the result of evaluating `AssignmentExpression`.
2. Let `arg` be `GetValue(ref)`.
3. Return a `List` whose sole item is `arg`.

ArgumentList : ... AssignmentExpression

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

ArgumentList : ArgumentList , AssignmentExpression

1. Let precedingArgs be ArgumentListEvaluation of ArgumentList.
2. ReturnIfAbrupt(precedingArgs).
3. Let ref be the result of evaluating AssignmentExpression.
4. Let arg be ? GetValue(ref).
5. Append arg to the end of precedingArgs.
6. Return precedingArgs.

ArgumentList : ArgumentList , ... AssignmentExpression

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

### 12.3.7 Tagged Templates

NOTE A tagged template is a function call where the arguments of the call are derived from a TemplateLiteral (12.2.9). The actual arguments include a template object (12.2.9.3) and the values produced by evaluating the expressions embedded within the TemplateLiteral.

### 12.3.7.1 Runtime Semantics: Evaluation

MemberExpression : MemberExpression TemplateLiteral

1. Let tagRef be the result of evaluating MemberExpression.
2. Let thisCall be this MemberExpression.
3. Let tailCall be IsInTailPosition(thisCall).
4. Return ? EvaluateCall(tagRef, TemplateLiteral, tailCall).

CallExpression : CallExpression TemplateLiteral
1. Let \( \text{tagRef} \) be the result of evaluating \( \text{CallExpression} \).
2. Let \( \text{thisCall} \) be this \( \text{CallExpression} \).
3. Let \( \text{tailCall} \) be \( \text{IsInTailPosition(\text{thisCall})} \).
4. Return \( \text{? EvaluateCall(\text{tagRef}, \text{TemplateLiteral, \text{tailCall}})} \).

### 12.3.8 Meta Properties

#### 12.3.8.1 Runtime Semantics: Evaluation

**NewTarget : new . target**

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

### 12.4 Update Expressions

**Syntax**

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

#### 12.4.1 Static Semantics: Early Errors

\[
\text{UpdateExpression} : \\
\quad \text{LeftHandSideExpression} ++ \\
\quad \text{LeftHandSideExpression} --
\]

- It is an early \( \text{Reference Error} \) if \( \text{IsValidSimpleAssignmentTarget} \) of \( \text{LeftHandSideExpression} \) is \text{false}.

\[
\text{UpdateExpression} : \\
\quad ++ \text{UnaryExpression} \\
\quad -- \text{UnaryExpression}
\]

- It is an early \( \text{Reference Error} \) if \( \text{IsValidSimpleAssignmentTarget} \) of \( \text{UnaryExpression} \) is \text{false}.

#### 12.4.2 Static Semantics: IsFunctionDefinition
UpdateExpression:
  LeftHandSideExpression ++
  LeftHandSideExpression --
  ++ UnaryExpression
  -- UnaryExpression

1. Return \texttt{false}.

12.4.3 Static Semantics: IsValidSimpleAssignmentTarget

UpdateExpression:
  LeftHandSideExpression ++
  LeftHandSideExpression --
  ++ UnaryExpression
  -- UnaryExpression

1. Return \texttt{false}.

12.4.4 Postfix Increment Operator

12.4.4.1 Runtime Semantics: Evaluation

\texttt{UpdateExpression : LeftHandSideExpression ++}

1. Let \texttt{lhs} be the result of evaluating \texttt{LeftHandSideExpression}.
2. Let \texttt{oldValue} be \texttt{ToNumber(? GetValue(lhs))}.
3. Let \texttt{newValue} be the result of adding the value 1 to \texttt{oldValue}, using the same rules as for the +
   operator (see 12.8.5).
4. Perform ? \texttt{PutValue(lhs, newValue)}.
5. Return \texttt{oldValue}.

12.4.5 Postfix Decrement Operator

12.4.5.1 Runtime Semantics: Evaluation

\texttt{UpdateExpression : LeftHandSideExpression --}

1. Let \texttt{lhs} be the result of evaluating \texttt{LeftHandSideExpression}.
2. Let \texttt{oldValue} be \texttt{ToNumber(? GetValue(lhs))}.
3. Let \texttt{newValue} be the result of subtracting the value 1 from \texttt{oldValue}, using the same rules as for the -
   operator (see 12.8.5).
4. Perform ? \texttt{PutValue(lhs, newValue)}. 

5. Return *oldValue*.

### 12.4.6 Prefix Increment Operator

#### 12.4.6.1 Runtime Semantics: Evaluation

*UpdateExpression* : `++ UnaryExpression`

1. Let `expr` be the result of evaluating `UnaryExpression`.
2. Let `oldValue` be `? ToNumber(? GetValue(expr))`.
3. Let `newValue` be the result of adding the value 1 to `oldValue`, using the same rules as for the `+` operator (see 12.8.5).
5. Return `newValue`.

### 12.4.7 Prefix Decrement Operator

#### 12.4.7.1 Runtime Semantics: Evaluation

*UpdateExpression* : `-- UnaryExpression`

1. Let `expr` be the result of evaluating `UnaryExpression`.
2. Let `oldValue` be `? ToNumber(? GetValue(expr))`.
3. Let `newValue` be the result of subtracting the value 1 from `oldValue`, using the same rules as for the `-` operator (see 12.8.5).
5. Return `newValue`.

### 12.5 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]
```
12.5.1 Static Semantics: IsFunctionDefinition

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

1. Return false.

12.5.2 Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return false.

12.5.3 The delete Operator

12.5.3.1 Static Semantics: Early Errors

UnaryExpression : delete UnaryExpression

- It is a Syntax Error if the UnaryExpression is contained in strict mode code and the derived UnaryExpression is PrimaryExpression : IdentifierReference .
It is a Syntax Error if the derived *UnaryExpression* is
*PrimaryExpression* : *CoverParenthesizedExpressionAndArrowParameterList*
and *CoverParenthesizedExpressionAndArrowParameterList* ultimately derives a phrase that, if used in place of *UnaryExpression*, would produce a Syntax Error according to these rules. This rule is recursively applied.

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

### 12.5.3.2 Runtime Semantics: Evaluation

*UnaryExpression* : `delete` *UnaryExpression*

1. Let `ref` be the result of evaluating *UnaryExpression*.
2. Return `IfAbrupt(ref)`.
3. If `Type(ref)` is not `Reference`, return `true`.
4. If `IsUnresolvableReference(ref)` is `true`, then
   a. Assert: `IsStrictReference(ref)` is `false`.
   b. Return `true`.
5. If `IsPropertyReference(ref)` is `true`, then
   a. If `IsSuperReference(ref)` is `true`, throw a `ReferenceError` exception.
   b. Let `baseObj` be `ToObject(GetBase(ref))`.
   c. Let `deleteStatus` be `? baseObj.[[Delete]](GetReferencedName(ref))`.
   d. If `deleteStatus` is `false` and `IsStrictReference(ref)` is `true`, throw a `TypeError` exception.
   e. Return `deleteStatus`.
6. Else `ref` is a `Reference` to an `Environment Record` binding,
   a. Let `bindings` be `GetBase(ref)`.
   b. Return `? bindings.DeleteBinding(GetReferencedName(ref))`.

**NOTE**  When a `delete` operator occurs within `strict mode code`, a `SyntaxError` exception is thrown if its *UnaryExpression* is a direct reference to a variable, function argument, or function name. In addition, if a `delete` operator occurs within `strict mode code` and the property to be deleted has the attribute `{ [[Configurable]]: false }`, a `TypeError` exception is thrown.

### 12.5.4 The `void` Operator

#### 12.5.4.1 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.

### 12.5.5 The `typeof` Operator

#### 12.5.5.1 Runtime Semantics: Evaluation

```
UnaryExpression : `typeof` UnaryExpression
```

1. Let `val` be the result of evaluating `UnaryExpression`.
2. If `Type(val)` is `Reference`, then
   a. If `IsUnresolvableReference(val)` is `true`, return "undefined".
3. Set `val` to `? GetValue(val)`.
4. Return a String according to Table 35.

#### Table 35: `typeof` Operator Results

<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>Object (ordinary and does not implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (standard exotic and does not implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (implements [[Call]])</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>Object (non-standard exotic and does not implement [[Call]])</td>
<td>Implementation-defined. Must not be &quot;undefined&quot;, &quot;boolean&quot;, &quot;function&quot;, &quot;number&quot;, &quot;symbol&quot;, or &quot;string&quot;.</td>
</tr>
</tbody>
</table>

**NOTE** Implementations are discouraged from defining new `typeof` result values for non-standard exotic objects. If possible "object" should be used for such objects.
### 12.5.6 Unary + Operator

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

#### 12.5.6.1 Runtime Semantics: Evaluation

\( UnaryExpression : + UnaryExpression \)

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

### 12.5.7 Unary - Operator

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

#### 12.5.7.1 Runtime Semantics: Evaluation

\( UnaryExpression : - UnaryExpression \)

1. Let \( expr \) be the result of evaluating \( UnaryExpression \).
2. Let \( oldValue \) be \( ? \ ToNumber(\ ? \ GetValue(expr)) \).
3. If \( oldValue \) is \( NaN \), return \( NaN \).
4. Return the result of negating \( oldValue \); that is, compute a Number with the same magnitude but opposite sign.

### 12.5.8 Bitwise NOT Operator (~)

#### 12.5.8.1 Runtime Semantics: Evaluation

\( UnaryExpression : ~ UnaryExpression \)

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

### 12.5.9 Logical NOT Operator (~)

#### 12.5.9.1 Runtime Semantics: Evaluation

\( UnaryExpression : ~ UnaryExpression \)

1. Let \( expr \) be the result of evaluating \( UnaryExpression \).
2. Let \( oldValue \) be \( ToBoolean(\ ? \ GetValue(expr)) \).
3. If \texttt{oldValue} is \texttt{true}, return \texttt{false}.
4. Return \texttt{true}.

### 12.6 Exponentiation Operator

**Syntax**

\[
\begin{align*}
\text{ExponentiationExpression} & : \text{Yield, Await} \\
\text{UnaryExpression} & : \text{Yield, Await} \\
\text{UpdateExpression} & : \text{Yield, Await}
\end{align*}
\]

\[\text{UpdateExpression} \quad \ast \ast \quad \text{ExponentiationExpression}\]

### 12.6.1 Static Semantics: IsFunctionDefinition

\[
\begin{align*}
\text{ExponentiationExpression} & : \\
\text{UpdateExpression} & \quad \ast \ast \quad \text{ExponentiationExpression}
\end{align*}
\]

1. Return \texttt{false}.

### 12.6.2 Static Semantics: IsValidSimpleAssignmentTarget

\[
\begin{align*}
\text{ExponentiationExpression} & : \\
\text{UpdateExpression} & \quad \ast \ast \quad \text{ExponentiationExpression}
\end{align*}
\]

1. Return \texttt{false}.

### 12.6.3 Runtime Semantics: Evaluation

\[
\text{ExponentiationExpression} : \text{UpdateExpression} \quad \ast \ast \quad \text{ExponentiationExpression}
\]

1. Let \textit{left} be the result of evaluating \textit{UpdateExpression}.
2. Let \textit{leftValue} be \texttt{GetValue(left)}.
3. Let \textit{right} be the result of evaluating \textit{ExponentiationExpression}.
4. Let \textit{rightValue} be \texttt{GetValue(right)}.
5. Let \textit{base} be \texttt{ToNumber(leftValue)}.
6. Let \textit{exponent} be \texttt{ToNumber(rightValue)}.
7. Return the result of \texttt{Applying the \ast \ast operator} with \textit{base} and \textit{exponent} as specified in 12.6.4.

### 12.6.4 Applying the \ast \ast Operator

Returns an implementation-dependent approximation of the result of raising \textit{base} to the power \textit{exponent}.
• If `exponent` is `NaN`, the result is `NaN`.
• If `exponent` is `+0`, the result is 1, even if `base` is `NaN`.
• If `exponent` is `-0`, the result is 1, even if `base` is `NaN`.
• If `base` is `NaN` and `exponent` is nonzero, the result is `NaN`.
• If `abs(base) > 1` and `exponent` is `+∞`, the result is `+∞`.
• If `abs(base) > 1` and `exponent` is `-∞`, the result is `+0`.
• If `abs(base)` is 1 and `exponent` is `+∞`, the result is `NaN`.
• If `abs(base)` is 1 and `exponent` is `-∞`, the result is `NaN`.
• If `abs(base)` < 1 and `exponent` is `+∞`, the result is `+0`.
• If `abs(base)` < 1 and `exponent` is `-∞`, the result is `+∞`.
• If `base` is `+∞` and `exponent` > 0, the result is `+∞`.
• If `base` is `+∞` and `exponent` < 0, the result is `+0`.
• If `base` is `-∞` and `exponent` > 0 and `exponent` is an odd integer, the result is `-∞`.
• If `base` is `-∞` and `exponent` > 0 and `exponent` is not an odd integer, the result is `+∞`.
• If `base` is `-∞` and `exponent` < 0 and `exponent` is an odd integer, the result is `-0`.
• If `base` is `-∞` and `exponent` < 0 and `exponent` is not an odd integer, the result is `+0`.
• If `base` is `-0` and `exponent` > 0, the result is `-0`.
• If `base` is `-0` and `exponent` < 0, the result is `-∞`.
• If `base` is `-0` and `exponent` < 0 and `exponent` is not an odd integer, the result is `+∞`.
• If `base` is `-0` and `exponent` < 0 and `exponent` is not an odd integer, the result is `NaN`.

**NOTE**  
The result of `base ** exponent` when `base` is 1 or `-1` and `exponent` is `+Infinity` or `-Infinity` differs from IEEE 754-2008. The first edition of ECMAScript specified a result of `NaN` for this operation, whereas later versions of IEEE 754-2008 specified 1. The historical ECMAScript behaviour is preserved for compatibility reasons.

### 12.7 Multiplicative Operators

**Syntax**

```plaintext
MultiplicativeExpression[Yield, Await] :  
  ExponentiationExpression[Yield, Await]  
  MultiplicativeExpression[Yield, Await]  
  MultiplicativeOperator

MultiplicativeOperator : one of
  * / %
```
### 12.7.1 Static Semantics: IsFunctionDefinition

*FunctionName* : *FunctionName* *FunctionBody*

1. Return **false**.

### 12.7.2 Static Semantics: IsValidSimpleAssignmentTarget

*FunctionName* : *FunctionName* *FunctionBody*

1. Return **false**.

### 12.7.3 Runtime Semantics: Evaluation

*FunctionName* : *FunctionName* *FunctionBody*

1. Let *left* be the result of evaluating *FunctionName*.
2. Let *leftValue* be ? `GetValue(left)`.
3. Let *right* be the result of evaluating *FunctionName*.
4. Let *rightValue* be ? `GetValue(right)`.
5. Let *inum* be ? `ToNumber(leftValue)`.
6. Let *rnum* be ? `ToNumber(rightValue)`.
7. Return the result of applying the *MultiplicativeOperator* (*, /, or %) to *inum* and *rnum* as specified in 12.7.3.1, 12.7.3.2, or 12.7.3.3.

#### 12.7.3.1 Applying the * MultiplicativeOperator

The *MultiplicativeOperator* performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in ECMAScript, because of finite precision.

The result of a floating-point multiplication is governed by the rules of IEEE 754-2008 binary double-precision arithmetic:

- If either operand is *NaN*, the result is *NaN*.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Multiplication of an infinity by a zero results in *NaN*.
- Multiplication of an infinity by an infinity results in an infinity. The sign is determined by the rule already stated above.
- Multiplication of an infinity by a finite nonzero value results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity nor *NaN* is involved, the product is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If the
magnitude is too small to represent, the result is then a zero of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754-2008.

12.7.3.2 Applying the / Operator

The / MultiplicativeOperator performs division, producing the quotient of its operands. The left operand is the dividend and the right operand is the divisor. ECMAScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification of IEEE 754-2008 arithmetic:

- If either operand is NaN, the result is NaN.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a zero results in an infinity. The sign is determined by the rule already stated above.
- Division of an infinity by a nonzero finite value results in a signed infinity. The sign is determined by the rule already stated above.
- Division of a finite value by an infinity results in zero. The sign is determined by the rule already stated above.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.
- Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode. If the magnitude is too large to represent, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754-2008.

12.7.3.3 Applying the % Operator

The % MultiplicativeOperator yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

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

The result of a floating-point remainder operation as computed by the % operator is not the same as the “remainder” operation defined by IEEE 754-2008. The IEEE 754-2008 “remainder” operation computes
the remainder from a rounding division, not a truncating division, and so its behaviour is not analogous to that of the usual integer remainder operator. Instead the ECMAScript language defines `%` on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function `fmod`.

The result of an ECMAScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is `NaN`, the result is `NaN`.
- The sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is `NaN`.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is nonzero and finite, the result is the same as the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor `NaN` is involved, the floating-point remainder `r` from a dividend `n` and a divisor `d` is defined by the mathematical relation `r = n - (d × q)` where `q` is an integer that is negative only if `n/d` is negative and positive only if `n/d` is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of `n` and `d`. `r` is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode.

### 12.8 Additive Operators

**Syntax**

```
AdditiveExpression[Yield, Await] :
  MultiplicativeExpression[Yield, Await]
AdditiveExpression[Yield, Await] + MultiplicativeExpression[Yield, Await]
AdditiveExpression[Yield, Await] - MultiplicativeExpression[Yield, Await]
```

### 12.8.1 Static Semantics: IsFunctionDefinition

```
AdditiveExpression :
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression
```

1. Return `false`.

### 12.8.2 Static Semantics: IsValidSimpleAssignmentTarget

```
AdditiveExpression :
```
12.8.3 The Addition Operator (+)

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

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + MultiplicativeExpression

1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. Let rref be the result of evaluating MultiplicativeExpression.
4. Let rval be GetValue(rref).
5. Let lprim be ToPrimitive(lval).
6. Let rprim be ToPrimitive(rval).
7. If Type(lprim) is String or Type(rprim) is String, then
   a. Let lstr be ToString(lprim).
   b. Let rstr be ToString(rprim).
   c. Return the String that is the result of concatenating lstr and rstr.
8. Let lnum be ToNumber(lprim).
9. Let rnum be ToNumber(rprim).
10. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.8.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 5 and 6. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 7 differs from step 3 of the Abstract Relational Comparison algorithm, by using the logical-or operation instead of the logical-and operation.

12.8.4 The Subtraction Operator (−)

12.8.4.1 Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression − MultiplicativeExpression

1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. Let `rref` be the result of evaluating `MultiplicativeExpression`.
4. Let `rval` be `GetValue(rref)`.
5. Let `lval` be `ToNumber(lval)`.
6. Let `rval` be `ToNumber(rval)`.
7. Return the result of applying the subtraction operation to `lval` and `rval`. See the note below 12.8.5.

### 12.8.5 Applying the Additive Operators to Numbers

The `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The `-` operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754-2008 binary double-precision arithmetic:

- If either operand is `NaN`, the result is `NaN`.
- The sum of two infinities of opposite sign is `NaN`.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two negative zeroes is `-0`. The sum of two positive zeroes, or of two zeroes of opposite sign, is `+0`.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is `+0`.
- In the remaining cases, where neither an infinity, nor a zero, nor `NaN` is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754-2008.

#### NOTE

The `-` operator performs subtraction when applied to two operands of numeric type, producing the difference of its operands; the left operand is the minuend and the right operand is the subtrahend. Given numeric operands `a` and `b`, it is always the case that `a - b` produces the same result as `a + (-b)`.

### 12.9 Bitwise Shift Operators

#### Syntax

```
ShiftExpression [Yield, Await] :
  AdditiveExpression [?Yield, ?Await]
```
12.9.1 Static Semantics: IsFunctionDefinition

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

1. Return false.

12.9.2 Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return false.

12.9.3 The Left Shift Operator ( << )

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

12.9.3.1 Runtime Semantics: Evaluation

ShiftExpression : ShiftExpression << AdditiveExpression

1. Let lref be the result of evaluating ShiftExpression.
2. Let lval be ? GetValue(lref).
3. Let rref be the result of evaluating AdditiveExpression.
4. Let rval be ? GetValue(rref).
5. Let lnum be ? ToInt32(lval).
7. Let shiftCount be the result of masking out all but the least significant 5 bits of rnum, that is, compute rnum & 0x1F.
8. Return the result of left shifting lnum by shiftCount bits. The result is a signed 32-bit integer.
12.9.4 The Signed Right Shift Operator (\( \gg \))

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

12.9.4.1 Runtime Semantics: Evaluation

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

1. Let \( lref \) be the result of evaluating \( \text{ShiftExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{AdditiveExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
5. Let \( lnum \) be \( \text{ToInt32}(lval) \).
6. Let \( rnum \) be \( \text{ToUint32}(rval) \).
7. Let \( shiftCount \) be the result of masking out all but the least significant 5 bits of \( rnum \), that is, compute \( rnum \& 0x1F \).
8. Return the result of performing a sign-extending right shift of \( lnum \) by \( shiftCount \) bits. The most significant bit is propagated. The result is a signed 32-bit integer.

12.9.5 The Unsigned Right Shift Operator (\( \gg\gg \))

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

12.9.5.1 Runtime Semantics: Evaluation

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

1. Let \( lref \) be the result of evaluating \( \text{ShiftExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. Let \( rref \) be the result of evaluating \( \text{AdditiveExpression} \).
4. Let \( rval \) be \( \text{GetValue}(rref) \).
5. Let \( lnum \) be \( \text{ToInt32}(lval) \).
6. Let \( rnum \) be \( \text{ToUint32}(rval) \).
7. Let \( shiftCount \) be the result of masking out all but the least significant 5 bits of \( rnum \), that is, compute \( rnum \& 0x1F \).
8. Return the result of performing a zero-filling right shift of \( lnum \) by \( shiftCount \) bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

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

Syntax

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

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} < \text{ShiftExpression}\{\text{Yield, Await}\}
\]

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} > \text{ShiftExpression}\{\text{Yield, Await}\}
\]

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} \leq \text{ShiftExpression}\{\text{Yield, Await}\}
\]

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} \geq \text{ShiftExpression}\{\text{Yield, Await}\}
\]

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} \text{ instanceof } \text{ShiftExpression}\{\text{Yield, Await}\}
\]

\[
\text{RelationalExpression}\{\text{In, Yield, Await}\} \text{ in } \text{ShiftExpression}\{\text{Yield, Await}\}
\]

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

12.10.1 Static Semantics: IsFunctionDefinition

\[
\text{RelationalExpression} : \\
\text{RelationalExpression} < \text{ShiftExpression} \\
\text{RelationalExpression} > \text{ShiftExpression} \\
\text{RelationalExpression} \leq \text{ShiftExpression} \\
\text{RelationalExpression} \geq \text{ShiftExpression} \\
\text{RelationalExpression} \text{ instanceof } \text{ShiftExpression} \\
\text{RelationalExpression} \text{ in } \text{ShiftExpression}
\]

1. Return false.

12.10.2 Static Semantics: IsValidSimpleAssignmentTarget

\[
\text{RelationalExpression} : \\
\text{RelationalExpression} < \text{ShiftExpression} \\
\text{RelationalExpression} > \text{ShiftExpression} \\
\text{RelationalExpression} \leq \text{ShiftExpression} \\
\text{RelationalExpression} \geq \text{ShiftExpression} \\
\text{RelationalExpression} \text{ instanceof } \text{ShiftExpression} \\
\text{RelationalExpression} \text{ in } \text{ShiftExpression}
\]
1. Return \texttt{false}.

### 12.10.3 Runtime Semantics: Evaluation

\texttt{RelationalExpression} : \texttt{RelationalExpression} \texttt{<} \texttt{ShiftExpression}

1. Let \texttt{lref} be the result of evaluating \texttt{RelationalExpression}.
2. Let \texttt{lval} be \texttt{GetValue(lref)}.
3. Let \texttt{rref} be the result of evaluating \texttt{ShiftExpression}.
4. Let \texttt{rval} be \texttt{GetValue(rref)}.
5. Let \texttt{r} be the result of performing \texttt{Abstract Relational Comparison \ lval < \ rval}.
6. ReturnIfAbrupt(\texttt{r}).
7. If \texttt{r} is \texttt{undefined}, return \texttt{false}. Otherwise, return \texttt{r}.

\texttt{RelationalExpression} : \texttt{RelationalExpression} \texttt{>} \texttt{ShiftExpression}

1. Let \texttt{lref} be the result of evaluating \texttt{RelationalExpression}.
2. Let \texttt{lval} be \texttt{GetValue(lref)}.
3. Let \texttt{rref} be the result of evaluating \texttt{ShiftExpression}.
4. Let \texttt{rval} be \texttt{GetValue(rref)}.
5. Let \texttt{r} be the result of performing \texttt{Abstract Relational Comparison \ rval < \ lval with LeftFirst equal to false}.
6. ReturnIfAbrupt(\texttt{r}).
7. If \texttt{r} is \texttt{undefined}, return \texttt{false}. Otherwise, return \texttt{r}.

\texttt{RelationalExpression} : \texttt{RelationalExpression} \texttt{<=} \texttt{ShiftExpression}

1. Let \texttt{lref} be the result of evaluating \texttt{RelationalExpression}.
2. Let \texttt{lval} be \texttt{GetValue(lref)}.
3. Let \texttt{rref} be the result of evaluating \texttt{ShiftExpression}.
4. Let \texttt{rval} be \texttt{GetValue(rref)}.
5. Let \texttt{r} be the result of performing \texttt{Abstract Relational Comparison \ rval < \ lval with LeftFirst equal to false}.
6. ReturnIfAbrupt(\texttt{r}).
7. If \texttt{r} is \texttt{true} or \texttt{undefined}, return \texttt{false}. Otherwise, return \texttt{true}.

\texttt{RelationalExpression} : \texttt{RelationalExpression} \texttt{>=} \texttt{ShiftExpression}

1. Let \texttt{lref} be the result of evaluating \texttt{RelationalExpression}.
2. Let \texttt{lval} be \texttt{GetValue(lref)}.
3. Let \texttt{rref} be the result of evaluating \texttt{ShiftExpression}.
4. Let \texttt{rval} be \texttt{GetValue(rref)}.
5. Let \texttt{r} be the result of performing \texttt{Abstract Relational Comparison \ lval < \ rval}. 

12.10.3 Runtime Semantics: Evaluation
6. ReturnIfAbrupt(r).
7. If r is true or undefined, return false. Otherwise, return true.

RelationalExpression : RelationalExpression instanceof ShiftExpression

1. Let lref be the result of evaluating RelationalExpression.
2. Let lval be ? GetValue(lref).
3. Let rref be the result of evaluating ShiftExpression.
4. Let rval be ? GetValue(rref).

RelationalExpression : RelationalExpression in ShiftExpression

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

12.10.4 Runtime Semantics: InstanceofOperator ( O, C )

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

1. If Type(C) is not Object, throw a TypeError exception.
2. Let instOfHandler be ? GetMethod(C, @@hasInstance).
3. If instOfHandler is not undefined, then
   a. Return ToBoolean(? Call(instOfHandler, C, « O »)).
4. If IsCallable(C) is false, throw a TypeError exception.

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 a function object does not define or inherit @@hasInstance it uses the default instanceof semantics.

12.11 Equality Operators

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

Syntax

EqualityExpression[In, Yield, Await] :
  RelationalExpression[?In, ?Yield, ?Await]
EqualityExpression[?In, ?Yield, ?Await] ===
  RelationalExpression[?In, ?Yield, ?Await]
EqualityExpression[?In, ?Yield, ?Await] !==
  RelationalExpression[?In, ?Yield, ?Await]

12.11.1 Static Semantics: IsFunctionDefinition

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

1. Return false.

12.11.2 Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return false.

12.11.3 Runtime Semantics: Evaluation

EqualityExpression : EqualityExpression == RelationalExpression

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

$equalityExpression : equalityExpression != relationalExpression$

1. Let $lref$ be the result of evaluating $equalityExpression$.
2. Let $lval$ be ? $GetValue(lref)$.
3. Let $rref$ be the result of evaluating $relationalExpression$.
4. Let $rval$ be ? $GetValue(rref)$.
5. Let $r$ be the result of performing Abstract Equality Comparison $rval == lval$.
6. If $r$ is true, return false. Otherwise, return true.

$equalityExpression : equalityExpression === relationalExpression$

1. Let $lref$ be the result of evaluating $equalityExpression$.
2. Let $lval$ be ? $GetValue(lref)$.
3. Let $rref$ be the result of evaluating $relationalExpression$.
4. Let $rval$ be ? $GetValue(rref)$.
5. Return the result of performing Strict Equality Comparison $rval === lval$.

$equalityExpression : equalityExpression !== relationalExpression$

1. Let $lref$ be the result of evaluating $equalityExpression$.
2. Let $lval$ be ? $GetValue(lref)$.
3. Let $rref$ be the result of evaluating $relationalExpression$.
4. Let $rval$ be ? $GetValue(rref)$.
5. Let $r$ be the result of performing Strict Equality Comparison $rval === lval$.
6. If $r$ is true, return false. Otherwise, return true.

**NOTE 1**
Given the above definition of equality:

- String comparison can be forced by: '"' + $a$ == '"' + $b$.
- Numeric comparison can be forced by: +$a$ == +$b$.
- Boolean comparison can be forced by: !$a$ == !$b$.

**NOTE 2**
The equality operators maintain the following invariants:

- $A != B$ is equivalent to !(A == B).
- $A == B$ is equivalent to $B == A$, except in the order of evaluation of $A$ and $B$.

**NOTE 3**
The equality operator is not always transitive. For example, there might be two distinct String objects, each representing the same String value; each String object would be considered equal to the String value by the == operator, but the two String objects would not be equal to each other. For example:
• `new String("a") == "a"` and `"a" == new String("a")` are both true.
• `new String("a") == new String("a")` is false.

**NOTE 4** Comparison of Strings uses a simple equality test on sequences of code unit values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore Strings values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalized form.

### 12.12 Binary Bitwise Operators

**Syntax**

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

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

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

**12.12.1 Static Semantics: IsFunctionDefinition**

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

1. Return `false`.

**12.12.2 Static Semantics: IsValidSimpleAssignmentTarget**

BitwiseANDExpression : BitwiseANDExpression & EqualityExpression
BitwiseXORExpression : BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression : BitwiseORExpression | BitwiseXORExpression
1. Return \textit{false}.

12.12.3 Runtime Semantics: Evaluation

The production $A : A @ B$, where $@$ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Let $lref$ be the result of evaluating $A$.
2. Let $lval$ be $\text{GetValue}(lref)$.
3. Let $rref$ be the result of evaluating $B$.
4. Let $rval$ be $\text{GetValue}(rref)$.
5. Let $lnum$ be $\text{ToInt32}(lval)$.
6. Let $rnum$ be $\text{ToInt32}(rval)$.
7. Return the result of applying the bitwise operator $@$ to $lnum$ and $rnum$. The result is a signed 32-bit integer.

12.13 Binary Logical Operators

Syntax

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

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

\textbf{NOTE} \quad \text{The value produced by a $\&\&$ or $||$ operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.}

12.13.1 Static Semantics: IsFunctionDefinition

\[
\text{LogicalANDExpression : LogicalANDExpression} \ & \& \ \text{BitwiseORExpression} \\
\text{LogicalORExpression : LogicalORExpression} \ | \ | \ \text{LogicalANDExpression}
\]

1. Return \textit{false}.
12.13.2 Static Semantics: IsValidSimpleAssignmentTarget

LogicalANDExpression : LogicalANDExpression && BitwiseORExpression

LogicalORExpression : LogicalORExpression || LogicalANDExpression

1. Return false.

12.13.3 Runtime Semantics: Evaluation

LogicalANDExpression : LogicalANDExpression && BitwiseORExpression

1. Let lref be the result of evaluating LogicalANDExpression.
2. Let lval be GetValue(lref).
3. Let lbool be ToBoolean(lval).
4. 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.


Syntax

ConditionalExpression[In, Yield, Await] :

LogicalORExpression[?In, ?Yield, ?Await]
LogicalORExpression[?In, ?Yield, ?Await] ?

AssignmentExpression[+In, ?Yield, ?Await] :
AssignmentExpression[?In, ?Yield, ?Await]

NOTE The grammar for a ConditionalExpression in ECMAScript is slightly different from that in C and Java, which each allow the second subexpression to be an Expression but restrict the third expression to be a ConditionalExpression. The motivation for this difference in ECMAScript is to allow an assignment expression to be governed by either arm of a
conditional and to eliminate the confusing and fairly useless case of a comma expression as the centre expression.

12.14.1 Static Semantics: IsFunctionDefinition
\[\text{ConditionalExpression} : \text{LogicalORExpression} \ ? \ \text{AssignmentExpression} : \text{AssignmentExpression}\]

1. Return \text{false}.

12.14.2 Static Semantics: IsValidSimpleAssignmentTarget
\[\text{ConditionalExpression} : \text{LogicalORExpression} \ ? \ \text{AssignmentExpression} : \text{AssignmentExpression}\]

1. Return \text{false}.

12.14.3 Runtime Semantics: Evaluation
\[\text{ConditionalExpression} : \text{LogicalORExpression} \ ? \ \text{AssignmentExpression} : \text{AssignmentExpression}\]

1. Let \text{lref} be the result of evaluating \text{LogicalORExpression}.
2. Let \text{lval} be \text{ToBoolean}(\text{GetValue(lref)}).
3. If \text{lval} is \text{true}, then
   a. Let \text{trueRef} be the result of evaluating the first \text{AssignmentExpression}.
   b. Return \text{GetValue(trueRef)}.
4. Else,
   a. Let \text{falseRef} be the result of evaluating the second \text{AssignmentExpression}.
   b. Return \text{GetValue(falseRef)}.

12.15 Assignment Operators

Syntax
\[
\text{AssignmentExpression} \ [\text{In, Yield, Await}] : \\
\quad \text{ConditionalExpression} \ [?\text{In}, ?\text{Yield}, ?\text{Await}] \\
\quad [?\text{Yield}] \ \text{YieldExpression} \ [?\text{In}, ?\text{Await}] \\
\quad \text{ArrowFunction} \ [?\text{In}, ?\text{Yield}, ?\text{Await}] \\
\quad \text{AsyncArrowFunction} \ [?\text{In}, ?\text{Yield}, ?\text{Await}] \\
\quad \text{LeftHandSideExpression} \ [?\text{Yield}, ?\text{Await}] = \text{AssignmentExpression} \ [?\text{In}, ?\text{Yield}, ?\text{Await}] \\
\quad \text{LeftHandSideExpression} \ [?\text{Yield}, ?\text{Await}] \ \text{AssignmentOperator} \\
\quad \text{AssignmentExpression} \ [?\text{In}, ?\text{Yield}, ?\text{Await}]
\]

\text{AssignmentOperator} : \text{one of}
### 12.15.1 Static Semantics: Early Errors

**AssignmentExpression : LeftHandSideExpression = AssignmentExpression**

- It is a Syntax Error if `LeftHandSideExpression` is either an `ObjectLiteral` or an `ArrayLiteral` and the lexical token sequence matched by `LeftHandSideExpression` cannot be parsed with no tokens left over using `AssignmentPattern` as the goal symbol.
- It is an early Reference Error if `LeftHandSideExpression` is neither an `ObjectLiteral` nor an `ArrayLiteral` and `IsValidSimpleAssignmentTarget` of `LeftHandSideExpression` is `false`.

**AssignmentExpression : LeftHandSideExpression AssignmentOperator AssignmentExpression**

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

### 12.15.2 Static Semantics: IsFunctionDefinition

**AssignmentExpression :**

- `ArrowFunction`
- `AsyncArrowFunction`

1. Return `true`.

**AssignmentExpression :**

- `YieldExpression`
  - `LeftHandSideExpression = AssignmentExpression`
  - `LeftHandSideExpression AssignmentOperator AssignmentExpression`

1. Return `false`.

### 12.15.3 Static Semantics: IsValidSimpleAssignmentTarget

**AssignmentExpression :**

- `YieldExpression`
- `ArrowFunction`
- `AsyncArrowFunction`
- `LeftHandSideExpression = AssignmentExpression`
- `LeftHandSideExpression AssignmentOperator AssignmentExpression`

1. Return `false`.
12.15.4 Runtime Semantics: Evaluation

AssignmentExpression : LeftHandSideExpression = AssignmentExpression

1. If LeftHandSideExpression is neither an ObjectLiteral nor an ArrayLiteral, then
   a. Let lref be the result of evaluating LeftHandSideExpression.
   b. ReturnIfAbrupt(lref).
   c. Let rref be the result of evaluating AssignmentExpression.
   d. Let rval be ? GetValue(rref).
   e. If IsAnonymousFunctionDefinition(AssignmentExpression) and IsIdentifierRef of LeftHandSideExpression are both true, then
      i. Let hasNameProperty be ? HasOwnProperty(rval, "name").
      ii. If hasNameProperty is false, perform SetFunctionName(rval, GetReferencedName(lref)).
   g. Return rval.

2. Let assignmentPattern be the parse of the source text corresponding to LeftHandSideExpression using AssignmentPattern as the goal symbol with its [Yield] and [Await] parameters set to the values used when parsing LeftHandSideExpression.

3. Let rref be the result of evaluating AssignmentExpression.

4. Let rval be ? GetValue(rref).

5. Let status be the result of performing DestructuringAssignmentEvaluation of assignmentPattern using rval as the argument.

6. ReturnIfAbrupt(status).

7. Return rval.

AssignmentExpression : LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Let lref be the result of evaluating LeftHandSideExpression.

2. Let lvval be ? GetValue(lref).

3. Let rref be the result of evaluating AssignmentExpression.

4. Let rval be ? GetValue(rref).

5. Let op be the @ where AssignmentOperator is @=.

6. Let r be the result of applying op to lvval and rval as if evaluating the expression lvval op rval.


8. Return r.

NOTE When an assignment occurs within strict mode code, it is a runtime error if lref in step 1.f of the first algorithm or step 7 of the second algorithm it is an unresolvable reference. If it is, a ReferenceError exception is thrown. 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 for which the IsExtensible predicate returns the value false. In these cases a TypeError exception is thrown.
12.15.5 Destructuring Assignment

Supplemental Syntax

In certain circumstances when processing an instance of the production `AssignmentExpression : LeftHandSideExpression = AssignmentExpression` the following grammar is used to refine the interpretation of `LeftHandSideExpression`.

```
AssignmentPattern[Yield, Await] :  
  ObjectAssignmentPattern[?Yield, ?Await]
  ArrayAssignmentPattern[?Yield, ?Await]

ObjectAssignmentPattern[Yield, Await] :
  { }
  { AssignmentPropertyList[?Yield, ?Await] } 
  { AssignmentPropertyList[?Yield, ?Await] , }

ArrayAssignmentPattern[Yield, Await] :
  [ Elision_opt AssignmentRestElement[?Yield, ?Await] opt ]
    AssignmentRestElement[?Yield, ?Await] opt ]

AssignmentPropertyList[Yield, Await] :
  AssignmentProperty[?Yield, ?Await]

AssignmentElementList[Yield, Await] :
  AssignmentElisionElement[?Yield, ?Await]

AssignmentElisionElement[Yield, Await] :
  Elision_opt AssignmentElement[?Yield, ?Await]

AssignmentProperty[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]

12.15.5.1 Static Semantics: Early Errors
AssignmentProperty : IdentifierReference Initializer

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

DestructuringAssignmentTarget : LeftHandSideExpression

- It is a Syntax Error if LeftHandSideExpression is either an ObjectLiteral or an ArrayLiteral and if the lexical token sequence matched by LeftHandSideExpression cannot be parsed with no tokens left over using AssignmentPattern as the goal symbol.
- It is a Syntax Error if LeftHandSideExpression is neither an ObjectLiteral nor an ArrayLiteral and IsValidSimpleAssignmentTarget(LeftHandSideExpression) is false.

12.15.5.2 Runtime Semantics: DestructuringAssignmentEvaluation

With parameter value.

ObjectAssignmentPattern : { }

2. Return NormalCompletion(empty).

ObjectAssignmentPattern :
{ AssignmentPropertyList }
{ AssignmentPropertyList , }

2. Return the result of performing DestructuringAssignmentEvaluation for AssignmentPropertyList using value as the argument.

ArrayAssignmentPattern : [ ]

1. Let iterator be ? GetIterator(value).
2. Return ? IteratorClose(iterator, NormalCompletion(empty)).

ArrayAssignmentPattern : [ Elision ]

1. Let iterator be ? GetIterator(value).
2. Let iteratorRecord be Record {[[Iterator]]: iterator, [[Done]]: false}.
3. Let result be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
4. If iteratorRecord.([Done]) is false, return ? IteratorClose(iterator, result).
5. Return result.

ArrayAssignmentPattern : [ Elision AssignmentRestElement ]

1. Let iterator be ? GetIterator(value).
2. Let iteratorRecord be Record {[[Iterator]]: iterator, [[Done]]: false}.
3. If Elision is present, then
   a. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
   b. NOTE: In the event of an abrupt completion, the above IteratorDestructuringAssignmentEvaluation will always set iteratorRecord.([Done]) to true.
   c. ReturnIfAbrupt(status).
4. Let result be the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentRestElement with iteratorRecord as the argument.
5. If iteratorRecord.([Done]) is false, return ? IteratorClose(iterator, result).
6. Return result.

ArrayAssignmentPattern : [ AssignmentElementList ]

1. Let iterator be ? GetIterator(value).
2. Let iteratorRecord be Record {[[Iterator]]: iterator, [[Done]]: false}.
3. Let result be the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElementList using iteratorRecord as the argument.
4. If iteratorRecord.([Done]) is false, return ? IteratorClose(iterator, result).
5. Return result.

ArrayAssignmentPattern : [ AssignmentElementList , Elision AssignmentRestElement ]

1. Let iterator be ? GetIterator(value).
2. Let iteratorRecord be Record {[[Iterator]]: iterator, [[Done]]: false}.
3. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElementList using iteratorRecord as the argument.
4. If status is an abrupt completion, then
   a. If iteratorRecord.([Done]) is false, return ? IteratorClose(iterator, status).
   b. Return Completion(status).
5. If Elision is present, then
   a. Set status to the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
b. NOTE: In the event of an abrupt completion, the above
   IteratorDestructuringAssignmentEvaluation will always set iteratorRecord.[[Done]] to true.
   c. ReturnIfAbrupt(status).
6. If AssignmentRestElement is present, then
   a. Set status to the result of performing IteratorDestructuringAssignmentEvaluation of
      AssignmentRestElement with iteratorRecord as the argument.
7. If iteratorRecord.[[Done]] is false, return ? IteratorClose(iterator, status).
8. Return Completion(status).

AssignmentPropertyList : AssignmentPropertyList , AssignmentProperty

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

AssignmentProperty : IdentifierReference Initializer

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. Let defaultValue be the result of evaluating Initializer.
   b. Set v to ? GetValue(defaultValue).
   c. If IsAnonymousFunctionDefinition(Initializer) is true, then
      i. Let hasNameProperty be ? HasOwnProperty(v, "name").
      ii. If hasNameProperty is false, perform SetFunctionName(v, P).

AssignmentProperty : PropertyName : AssignmentElement

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

12.15.5.3 Runtime Semantics: IteratorDestructuringAssignmentEvaluation

With parameter iteratorRecord.

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

AssignmentElementList : AssignmentElementList , AssignmentElisionElement

1. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElementList using iteratorRecord as the argument.
2. ReturnIfAbrupt(status).
3. 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. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
2. ReturnIfAbrupt(status).
3. 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.[[Iterator]]).
   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. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
2. ReturnIfAbrupt(status).
3. If iteratorRecord.[[Done]] is false, then
   a. Let next be IteratorStep(iteratorRecord.[[Iterator]]).
   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.
4. Return NormalCompletion(empty).
AssignmentElement : DestructuringAssignmentTarget Initializer

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.[[Iterator]]).
   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. Let defaultValue be the result of evaluating Initializer.
   b. 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 parse of the source text corresponding to
      DestructuringAssignmentTarget using AssignmentPattern as the goal symbol with its [Yield] and
      [Await] parameters set to the values used when parsing this DestructuringAssignmentTarget.
   b. Return the result of performing DestructuringAssignmentEvaluation of
      nestedAssignmentPattern with v as the argument.
7. If Initializer is present and value is undefined and IsAnonymousFunctionDefinition(Initializer) and
   IsIdentifierRef of DestructuringAssignmentTarget are both true, then
   a. Let hasNameProperty be ? HasOwnProperty(v, "name").
   b. If hasNameProperty is false, perform SetFunctionName(v, GetReferencedName(lref)).

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 \( \text{iteratorRecord}.[[\text{Done}]] \) is \text{false},
   a. Let \( \text{next} \) be \( \text{IteratorStep}(\text{iteratorRecord}.[[\text{Iterator}]]). \)
   b. If \( \text{next} \) is an abrupt completion, set \( \text{iteratorRecord}.[[\text{Done}]] \) to \text{true}.
   c. \text{ReturnIfAbrupt}(\text{next}).
   d. If \( \text{next} \) is \text{false}, set \( \text{iteratorRecord}.[[\text{Done}]] \) to \text{true}.
   e. Else,
      i. Let \( \text{nextValue} \) be \( \text{IteratorValue}(\text{next}). \)
      ii. If \( \text{nextValue} \) is an abrupt completion, set \( \text{iteratorRecord}.[[\text{Done}]] \) to \text{true}.
      iii. \text{ReturnIfAbrupt}(\text{nextValue}).
      iv. Let \( \text{status} \) be \( \text{CreateDataProperty}(A, \text{ToString}(n), \text{nextValue}). \)
      v. Assert: \( \text{status} \) is \text{true}.
      vi. Increment \( n \) by 1.

5. If \( \text{DestructuringAssignmentTarget} \) is neither an \text{ObjectLiteral} nor an \text{ArrayLiteral}, then
   a. Return \( ? \text{PutValue}(\text{iref}, A). \)

6. Let \( \text{nestedAssignmentPattern} \) be the parse of the source text corresponding to
   \( \text{DestructuringAssignmentTarget} \) using \( \text{AssignmentPattern} \) as the \text{goal symbol} with its \text{[Yield]} and \text{[Await]} parameters set to the values used when parsing this \text{AssignmentRestElement}.

7. Return the result of performing \( \text{DestructuringAssignmentEvaluation} \) of \( \text{nestedAssignmentPattern} \) with \( A \) as the argument.

**12.15.5.4 Runtime Semantics: KeyedDestructuringAssignmentEvaluation**

With parameters \( \text{value} \) and \( \text{propertyName} \).

\( \text{AssignmentElement} : \text{DestructuringAssignmentTarget} \ \text{Initializer} \)

1. If \( \text{DestructuringAssignmentTarget} \) is neither an \text{ObjectLiteral} nor an \text{ArrayLiteral}, then
   a. Let \( \text{iref} \) be the result of evaluating \( \text{DestructuringAssignmentTarget}. \)
   b. \text{ReturnIfAbrupt}(\text{iref}).
2. Let \( v \) be \( ? \text{GetV}(\text{value}, \text{propertyName}). \)
3. If \( \text{Initializer} \) is present and \( v \) is \text{undefined}, then
   a. Let \( \text{defaultValue} \) be the result of evaluating \( \text{Initializer}. \)
   b. Let \( \text{rhsValue} \) be \( ? \text{GetValue}(\text{defaultValue}). \)
4. Else, let \( \text{rhsValue} \) be \( v. \)
5. If \( \text{DestructuringAssignmentTarget} \) is an \text{ObjectLiteral} or an \text{ArrayLiteral}, then
   a. Let \( \text{assignmentPattern} \) be the parse of the source text corresponding to
      \( \text{DestructuringAssignmentTarget} \) using \( \text{AssignmentPattern} \) as the \text{goal symbol} with its \text{[Yield]} and \text{[Await]} parameters set to the values used when parsing this \( \text{DestructuringAssignmentTarget}. \)
   b. Return the result of performing \( \text{DestructuringAssignmentEvaluation} \) of \( \text{assignmentPattern} \) with \( \text{rhsValue} \) as the argument.
6. If `Initializer` is present and `v` is `undefined` and `IsAnonymousFunctionDefinition(Initializer)` and `IsIdentifierRef` of `DestructuringAssignmentTarget` are both `true`, then
   a. Let `hasNameProperty` be `HasOwnProperty(rhsValue, "name")`.
   b. If `hasNameProperty` is `false`, perform `SetFunctionName(rhsValue, GetReferencedName(lref))`.

12.16 Comma Operator ( , )

Syntax

```
Expression[In, Yield, Await] :
  AssignmentExpression[?In, ?Yield, ?Await]
```

12.16.1 Static Semantics: IsFunctionDefinition

`Expression : Expression , AssignmentExpression`

1. Return `false`.

12.16.2 Static Semantics: IsValidSimpleAssignmentTarget

`Expression : Expression , AssignmentExpression`

1. Return `false`.

12.16.3 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.

13 ECMAScript Language: Statements and Declarations
Syntax

Statement[Yield, Await, Return] :
BlockStatement[?Yield, ?Await, ?Return]
VariableStatement[?Yield, ?Await]
EmptyStatement
ExpressionStatement[?Yield, ?Await]
IfStatement[?Yield, ?Await, ?Return]
ContinueStatement[?Yield, ?Await]
BreakStatement[?Yield, ?Await]
[+Return] ReturnStatement[?Yield, ?Await]
WithStatement[?Yield, ?Await, ?Return]
LabelledStatement[?Yield, ?Await, ?Return]
ThrowStatement[?Yield, ?Await]
TryStatement[?Yield, ?Await]
DebuggerStatement

Declaration[Yield, Await] :
HoistableDeclaration[?Yield, ?Await, ~Default]
ClassDeclaration[?Yield, ?Await, ~Default]
LexicalDeclaration[+In, ?Yield, ?Await]

HoistableDeclaration[Yield, Await, Default] :
FunctionDeclaration[?Yield, ?Await, ?Default]
GeneratorDeclaration[?Yield, ?Await, ?Default]
AsyncFunctionDeclaration[?Yield, ?Await, ?Default]

BreakableStatement[Yield, Await, Return] :
IterationStatement[?Yield, ?Await, ?Return]
SwitchStatement[?Yield, ?Await, ?Return]

13.1 Statement Semantics

13.1.1 Static Semantics: ContainsDuplicateLabels

With parameter labelSet.
Statement:
- VariableStatement
- EmptyStatement
- ExpressionStatement
- ContinueStatement
- BreakStatement
- ReturnStatement
- ThrowStatement
- DebuggerStatement

1. Return false.

### 13.1.2 Static Semantics: ContainsUndefinedBreakTarget

With parameter labelSet.

Statement:
- VariableStatement
- EmptyStatement
- ExpressionStatement
- ContinueStatement
- ReturnStatement
- ThrowStatement
- DebuggerStatement

1. Return false.

### 13.1.3 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.

Statement:
- VariableStatement
- EmptyStatement
- ExpressionStatement
- BreakStatement
- ReturnStatement
- ThrowStatement
- DebuggerStatement

1. Return false.
BreakableStatement : IterationStatement

1. Let newIterationSet be a copy of iterationSet with all the elements of labelSet appended.
2. Return ContainsUndefinedContinueTarget of IterationStatement with arguments newIterationSet and « ».

13.1.4 Static Semantics: DeclarationPart

HoistableDeclaration : FunctionDeclaration

1. Return FunctionDeclaration.

HoistableDeclaration : GeneratorDeclaration

1. Return GeneratorDeclaration.

HoistableDeclaration : AsyncFunctionDeclaration

1. Return AsyncFunctionDeclaration.

Declaration : ClassDeclaration

1. Return ClassDeclaration.

Declaration : LexicalDeclaration

1. Return LexicalDeclaration.

13.1.5 Static Semantics: VarDeclaredNames

Statement :
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement

1. Return a new empty List.

13.1.6 Static Semantics: VarScopedDeclarations

Statement :
1. Return a new empty List.

### 13.1.7 Runtime Semantics: LabelledEvaluation

With parameter `labelSet`.

**BreakableStatement : IterationStatement**

1. Let `stmtResult` be the result of performing LabelledEvaluation of `IterationStatement` with argument `labelSet`.
2. If `stmtResult`.[[Type]] is `break`, then
   a. If `stmtResult`.[[Target]] is empty, then
      i. If `stmtResult`.[[Value]] is empty, set `stmtResult` to `NormalCompletion(undefined)`.
      ii. Else, set `stmtResult` to `NormalCompletion(stmtResult`.[[Value]])`.
3. Return `Completion(stmtResult)`.

**BreakableStatement : SwitchStatement**

1. Let `stmtResult` be the result of evaluating `SwitchStatement`.
2. If `stmtResult`.[[Type]] is `break`, then
   a. If `stmtResult`.[[Target]] is empty, then
      i. If `stmtResult`.[[Value]] is empty, set `stmtResult` to `NormalCompletion(undefined)`.
      ii. Else, set `stmtResult` to `NormalCompletion(stmtResult`.[[Value]])`.
3. Return `Completion(stmtResult)`.

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

### 13.1.8 Runtime Semantics: Evaluation

**HoistableDeclaration : GeneratorDeclaration**

1. Return `NormalCompletion(empty)`.

**HoistableDeclaration : AsyncFunctionDeclaration**

1. Return `NormalCompletion(empty)`. 
HoistableDeclaration : FunctionDeclaration

1. Return the result of evaluating FunctionDeclaration.

BreakableStatement :
   IterationStatement
   SwitchStatement

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

13.2 Block

Syntax

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

 Block[Yield, Await, Return] :

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

 StatementListItem[Yield, Await, Return] :
    Statement[?Yield, ?Await, ?Return]
    Declaration[?Yield, ?Await]

13.2.1 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.

13.2.2 Static Semantics: ContainsDuplicateLabels

With parameter labelSet.
\begin{quote}
\textbf{Block} : \{ \}

1. Return \texttt{false}.

\textbf{StatementList} : \textbf{StatementList} \textbf{StatementListItem}

1. Let \texttt{hasDuplicates} be \texttt{ContainsDuplicateLabels of StatementList with argument labelSet}.
2. If \texttt{hasDuplicates} is \texttt{true}, return \texttt{true}.
3. Return \texttt{ContainsDuplicateLabels of StatementListItem with argument labelSet}.

\textbf{StatementListItem} : \textbf{Declaration}

1. Return \texttt{false}.

\subsection*{13.2.3 Static Semantics: \texttt{ContainsUndefinedBreakTarget}}

With parameter \texttt{labelSet}.

\textbf{Block} : \{ \}

1. Return \texttt{false}.

\textbf{StatementList} : \textbf{StatementList} \textbf{StatementListItem}

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

\textbf{StatementListItem} : \textbf{Declaration}

1. Return \texttt{false}.

\subsection*{13.2.4 Static Semantics: \texttt{ContainsUndefinedContinueTarget}}

With parameters \texttt{iterationSet} and \texttt{labelSet}.

\textbf{Block} : \{ \}

1. Return \texttt{false}.

\textbf{StatementList} : \textbf{StatementList} \textbf{StatementListItem}

1. Let \texttt{hasUndefinedLabels} be \texttt{ContainsUndefinedContinueTarget of StatementList with arguments iterationSet and \texttt{« »}}.
2. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
3. Return ContainsUndefinedContinueTarget of StatementListLItem with arguments iterationSet and « ».

StatementListLItem : Declaration

1. Return false.

13.2.5 Static Semantics: LexicallyDeclaredNames

Block : { }

1. Return a new empty List.

StatementList : StatementList StatementListLItem

1. Let names be LexicallyDeclaredNames of StatementList.
2. Append to names the elements of the LexicallyDeclaredNames of StatementListLItem.
3. Return names.

StatementListLItem : Statement

1. If Statement is Statement : LabeledStatement, return LexicallyDeclaredNames of LabeledStatement.
2. Return a new empty List.

StatementListLItem : Declaration

1. Return the BoundNames of Declaration.

13.2.6 Static Semantics: LexicallyScopedDeclarations

StatementList : StatementList StatementListLItem

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

StatementListLItem : Statement

1. If Statement is Statement : LabeledStatement, return LexicallyScopedDeclarations of LabeledStatement.
2. Return a new empty List.

StatementListLItem : Declaration

1. Return a new List containing DeclarationPart of Declaration.
13.2.7 Static Semantics: `TopLevelLexicallyDeclaredNames`

`StatementList : StatementList StatementListItem`

1. Let `names` be `TopLevelLexicallyDeclaredNames` of `StatementList`.
2. Append to `names` the elements of the `TopLevelLexicallyDeclaredNames` of `StatementListItem`.
3. Return `names`.

`StatementListItem : Statement`

1. Return a new empty `List`.

`StatementListItem : Declaration`

1. If `Declaration` is `Declaration : HoistableDeclaration`, then
   a. Return « ».
2. Return the `BoundNames` of `Declaration`.

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

13.2.8 Static Semantics: `TopLevelLexicallyScopedDeclarations`

`Block : { }`

1. Return a new empty `List`.

`StatementList : StatementList StatementListItem`

1. Let `declarations` be `TopLevelLexicallyScopedDeclarations` of `StatementList`.
2. Append to `declarations` the elements of the `TopLevelLexicallyScopedDeclarations` of `StatementListItem`.
3. Return `declarations`.

`StatementListItem : Statement`

1. Return a new empty `List`.

`StatementListItem : Declaration`

1. If `Declaration` is `Declaration : HoistableDeclaration`, then
   a. Return « ».
2. Return a new `List` containing `Declaration`.

13.2.9 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.

13.2.10 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.

### 13.2.11 Static Semantics: VarDeclaredNames

**Block : { }**

1. Return a new empty List.

**StatementList : StatementList StatementListItem**

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

**StatementListItem : Declaration**

1. Return a new empty List.

### 13.2.12 Static Semantics: VarScopedDeclarations

**Block : { }**

1. Return a new empty List.

**StatementList : StatementList StatementListItem**

1. Let *declarations* be VarScopedDeclarations of *StatementList*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *StatementListItem*.
3. Return *declarations*.

**StatementListItem : Declaration**

1. Return a new empty List.

### 13.2.13 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. Return `IfAbrupt(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;")
```

### 13.2.14 Runtime Semantics: 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.

BlockDeclarationInstantiation is performed as follows using arguments `code` and `env`. `code` is the Parse Node corresponding to the body of the block. `env` is the Lexical Environment in which bindings are to be created.

1. Let `envRec` be `env's` EnvironmentRecord.
2. Assert: `envRec` is a declarative Environment Record.
3. Let `declarations` be the LexicallyScopedDeclarations of `code`.
4. For each element `d` in `declarations`, do
   a. For each element `dn` of the BoundNames of `d`, do
      i. If `IsConstantDeclaration of d` is `true`, then
         1. Perform `! envRec.CreateImmutableBinding(dn, true)`.
      ii. Else,
         1. Perform `! envRec.CreateMutableBinding(dn, false)`.
   b. If `d` is a `FunctionDeclaration`, a `GeneratorDeclaration`, or an `AsyncFunctionDeclaration`, then
      i. Let `fn` be the sole element of the BoundNames of `d`. 
Let \( fo \) be the result of performing \( \text{InstantiateFunctionObject} \) for \( d \) with argument \( env \).

iii. Perform \( \text{envRec}.\text{InitializeBinding}(fn, fo) \).

### 13.3 Declarations and the Variable Statement

#### 13.3.1 Let and Const Declarations

**NOTE**  
\( \text{let} \) and \( \text{const} \) declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing LexicalEnvironment 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 \( \text{Initializer} \) is assigned the value of its \( \text{Initializer}'s \text{AssignmentExpression} \) when the LexicalBinding is evaluated, not when the variable is created. If a LexicalBinding in a \( \text{let} \) declaration does not have an \( \text{Initializer} \) the variable is assigned the value \( \text{undefined} \) when the LexicalBinding is evaluated.

**Syntax**

\[
\text{LexicalDeclaration}[\text{In, Yield, Await}] : \\
\text{LetOrConst BindingList}[^\text{In}, ^\text{Yield}, ^\text{Await}] ;
\]

\[
\text{LetOrConst} : \\
\text{let} \\
\text{const}
\]

\[
\text{BindingList}[\text{In, Yield, Await}] : \\
\text{LexicalBinding}[^\text{In}, ^\text{Yield}, ^\text{Await}] \\
\text{BindingList}[^\text{In}, ^\text{Yield}, ^\text{Await}] , \text{LexicalBinding}[^\text{In}, ^\text{Yield}, ^\text{Await}]
\]

\[
\text{LexicalBinding}[\text{In, Yield, Await}] : \\
\text{BindingIdentifier}[^\text{Yield}, ^\text{Await}] \text{Initializer}[^\text{In}, ^\text{Yield}, ^\text{Await}] \text{opt} \\
\text{BindingPattern}[^\text{Yield}, ^\text{Await}] \text{Initializer}[^\text{In}, ^\text{Yield}, ^\text{Await}]
\]

#### 13.3.1.1 Static Semantics: Early Errors

**LexicalDeclaration : LetOrConst BindingList ;**

- It is a Syntax Error if the BoundNames of \( \text{BindingList} \) contains "\text{let}".
- It is a Syntax Error if the BoundNames of \( \text{BindingList} \) contains any duplicate entries.

**LexicalBinding : BindingIdentifier \text{Initializer}**
• It is a Syntax Error if \textit{Initializer} is not present and \textit{IsConstantDeclaration} of the \textit{LexicalDeclaration} containing this \textit{LexicalBinding} is \texttt{true}.

\textbf{13.3.1.2 Static Semantics: BoundNames}

\textit{LexicalDeclaration} : \texttt{LetOrConst BindingList ;}

1. Return the BoundNames of \textit{BindingList}.

\textit{BindingList} : \textit{BindingList} , \textit{LexicalBinding}

1. Let \textit{name}s be the BoundNames of \textit{BindingList}.
2. Append to \textit{name}s the elements of the BoundNames of \textit{LexicalBinding}.
3. Return \textit{name}s.

\textit{LexicalBinding} : \textit{BindingIdentifier} \textit{Initializer}

1. Return the BoundNames of \textit{BindingIdentifier}.

\textit{LexicalBinding} : \textit{BindingPattern} \textit{Initializer}

1. Return the BoundNames of \textit{BindingPattern}.

\textbf{13.3.1.3 Static Semantics: IsConstantDeclaration}

\textit{LexicalDeclaration} : \texttt{LetOrConst BindingList ;}

1. Return \textit{IsConstantDeclaration} of \textit{LetOrConst}.

\textit{LetOrConst} : \texttt{let}

1. Return \texttt{false}.

\textit{LetOrConst} : \texttt{const}

1. Return \texttt{true}.

\textbf{13.3.1.4 Runtime Semantics: Evaluation}

\textit{LexicalDeclaration} : \texttt{LetOrConst BindingList ;}

1. Let \textit{next} be the result of evaluating \textit{BindingList}.
2. \texttt{ReturnIfAbrupt}({\textit{next}}).
3. Return \texttt{NormalCompletion}({\textit{empty}}).

\textit{BindingList} : \textit{BindingList} , \textit{LexicalBinding}

1. Let \textit{next} be the result of evaluating \textit{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. Let `rhs` be the result of evaluating `Initializer`.
4. Let `value` be `? GetValue(rhs)`.
5. If `IsAnonymousFunctionDefinition(Initializer)` is `true`, then
   a. Let `hasNameProperty` be `? HasOwnProperty(value, "name")`.
   b. If `hasNameProperty` is `false`, perform `SetFunctionName(value, bindingId)`.
6. 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.

### 13.3.2 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 `Lexical Environment` is instantiated and are initialized to `undefined` when created. Within the scope of any VariableEnvironment a common `BindingIdentifier` may appear in more than one `VariableDeclaration` but those declarations collective define only one variable. A variable defined by a `VariableDeclaration` with an `Initializer` is assigned the value of its `Initializer`'s `AssignmentExpression` when the `VariableDeclaration` is executed, not when the variable is created.
VariableStatement[Yield, Await] : 
    var VariableDeclarationList[+In, ?Yield, ?Await] ;

VariableDeclarationList[In, Yield, Await] : 
    VariableDeclaration[?In, ?Yield, ?Await] 
    VariableDeclarationList[?In, ?Yield, ?Await] 
    VariableDeclaration[?In, ?Yield, ?Await]

VariableDeclaration[In, Yield, Await] : 

13.3.2.1 Static Semantics: BoundNames
VariableDeclarationList : VariableDeclarationList , VariableDeclaration

1. Let names be BoundNames of VariableDeclarationList.
2. Append to names the elements of BoundNames of VariableDeclaration.
3. Return names.

VariableDeclaration : BindingIdentifier Initializer

1. Return the BoundNames of BindingIdentifier.

VariableDeclaration : BindingPattern Initializer

1. Return the BoundNames of BindingPattern.

13.3.2.2 Static Semantics: VarDeclaredNames
VariableStatement : var VariableDeclarationList ;

1. Return BoundNames of VariableDeclarationList.

13.3.2.3 Static Semantics: VarScopedDeclarations
VariableDeclarationList : VariableDeclaration

1. Return a new List containing VariableDeclaration.

VariableDeclarationList : VariableDeclarationList , VariableDeclaration

1. Let declarations be VarScopedDeclarations of VariableDeclarationList.
2. Append VariableDeclaration to declarations.
3. Return declarations.
13.3.2.4 Runtime Semantics: Evaluation

\[ \text{VariableStatement : var VariableDeclarationList ;} \]

1. Let \( next \) be the result of evaluating \( \text{VariableDeclarationList} \).
2. ReturnIfAbrupt(\( next \)).
3. Return \( \text{NormalCompletion}(\text{empty}) \).

\[ \text{VariableDeclarationList : VariableDeclarationList , VariableDeclaration} \]

1. Let \( next \) be the result of evaluating \( \text{VariableDeclarationList} \).
2. ReturnIfAbrupt(\( next \)).
3. Return the result of evaluating \( \text{VariableDeclaration} \).

\[ \text{VariableDeclaration} : BindingIdentifier \]

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

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

1. Let \( bindingId \) be \( \text{StringValue of BindingIdentifier} \).
2. Let \( lhs \) be \( ? \text{ResolveBinding}(bindingId) \).
3. Let \( rhs \) be the result of evaluating \( \text{Initializer} \).
4. Let \( value \) be \( ? \text{GetValue}(rhs) \).
5. If \( \text{IsAnonymousFunctionDefinition}(\text{Initializer}) \) is \( \text{true} \), then
   a. Let \( hasNameProperty \) be \( ? \text{HasOwnProperty}(value, "name") \).
   b. If \( hasNameProperty \) is \( \text{false} \), perform \( \text{SetFunctionName}(value, bindingId) \).
6. Return \( ? \text{PutValue}(lhs, value) \).

\[ \text{NOTE} \]

If a \( \text{VariableDeclaration} \) is nested within a \text{with} statement and the \( \text{BindingIdentifier} \) in the \( \text{VariableDeclaration} \) is the same as a property name of the binding object of the \text{with} statement’s object \( \text{Environment Record} \), then step 6 will assign \( value \) to the property instead of assigning to the VariableEnvironment binding of the \( \text{Identifier} \).

\[ \text{VariableDeclaration} : BindingPattern Initializer \]

1. Let \( rhs \) be the result of evaluating \( \text{Initializer} \).
2. Let \( rval \) be \( ? \text{GetValue}(rhs) \).
3. Return the result of performing BindingInitialization for \( \text{BindingPattern} \) passing \( rval \) and \( \text{undefined} \) as arguments.

13.3.3 Destructuring Binding Patterns

Syntax
**BindingPattern** [Yield, Await] :

  **ObjectBindingPattern** [?Yield, ?Await]

  **ArrayBindingPattern** [?Yield, ?Await]

**ObjectBindingPattern** [Yield, Await] :

  
  
  
  

  

**ArrayBindingPattern** [Yield, Await] :

  [ Elision\_opt  **BindingRestElement** [?Yield, ?Await] \_opt  ]

  [ **BindingElementList** [?Yield, ?Await] ]

  [ **BindingElementList** [?Yield, ?Await] , Elision\_opt  **BindingRestElement** [?Yield, ?Await] \_opt  ]

**BindingPropertyList** [Yield, Await] :

  **BindingProperty** [?Yield, ?Await]

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

**BindingElementList** [Yield, Await] :

  **BindingElisionElement** [?Yield, ?Await]

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

**BindingElisionElement** [Yield, Await] :

  Elision\_opt  **BindingElement** [?Yield, ?Await]

**BindingProperty** [Yield, Await] :

  **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] :

  ... **BindingIdentifier** [?Yield, ?Await]

  ... **BindingPattern** [?Yield, ?Await]
13.3.3.1 Static Semantics: BoundNames

ObjectBindingPattern : {  }

1. Return a new empty List.

ArrayBindingPattern : [  Elision  ]

1. Return a new empty List.

ArrayBindingPattern : [  Elision  BindingRestElement  ]

1. Return the BoundNames of BindingRestElement.

ArrayBindingPattern : [  BindingElementList ,  Elision  ]

1. Return the BoundNames of BindingElementList.

ArrayBindingPattern : [  BindingElementList ,  Elision  BindingRestElement  ]

1. Let names be BoundNames of BindingElementList.
2. Append to names the elements of BoundNames of BindingRestElement.
3. Return names.

BindingPropertyList : BindingPropertyList ,  BindingProperty

1. Let names be BoundNames of BindingPropertyList.
2. Append to names the elements of BoundNames of BindingProperty.
3. Return names.

BindingElementList : BindingElementList ,  BindingElisionElement

1. Let names be BoundNames of BindingElementList.
2. Append to names the elements of BoundNames of BindingElisionElement.
3. Return names.

BindingElisionElement : Elision  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.

13.3.3.2 Static Semantics: ContainsExpression

ObjectBindingPattern : { }

1. Return false.

ArrayBindingPattern : [ Elision ]

1. Return false.

ArrayBindingPattern : [ Elision BindingRestElement ]

1. Return ContainsExpression of BindingRestElement.

ArrayBindingPattern : [ BindingElementList , Elision ]

1. Return ContainsExpression of BindingElementList.

ArrayBindingPattern : [ BindingElementList , Elision 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 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.

13.3.3.3 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.

13.3.3.4 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.

### 13.3.3.5 Runtime Semantics: BindingInitialization

With parameters _value_ and _environment_.

**NOTE** When `undefined` is passed for _environment_ it indicates that a `PutValue` operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

_BindingPattern : ObjectBindingPattern_

1. Perform `RequireObjectCoercible(value).`
2. Return the result of performing BindingInitialization for _ObjectBindingPattern_ using _value_ and _environment_ as arguments.

_BindingPattern : ArrayBindingPattern_

1. Let _iterator_ be `GetIterator(value).`
2. Let _iteratorRecord_ be `Record{[[Iterator]]: iterator, [[Done]]: false}.`
3. Let _result_ be IteratorBindingInitialization for _ArrayBindingPattern_ using _iteratorRecord_ and _environment_ as arguments.
4. If _iteratorRecord._[[Done]]_ is _false_, return `IteratorClose(iterator, result).`
5. Return _result_.

_ObjectBindingPattern : {}_

1. Return `NormalCompletion(empty).`

_BindingPropertyList : BindingPropertyList , BindingProperty_

1. Let _status_ be the result of performing BindingInitialization for _BindingPropertyList_ using _value_ and _environment_ as arguments.
2. Return `IfAbrupt(status).`
3. Return the result of performing BindingInitialization for _BindingProperty_ using _value_ and _environment_ as arguments.
BindingProperty : SingleNameBinding

1. Let name be the string that is the only element of BoundNames of SingleNameBinding.
2. Return the result of performing KeyedBindingInitialization for SingleNameBinding using value, environment, and name as the arguments.

BindingProperty : PropertyName : BindingElement

1. Let P be the result of evaluating PropertyName.
2. ReturnIfAbrupt(P).
3. Return the result of performing KeyedBindingInitialization for BindingElement using value, environment, and P as arguments.

13.3.3.6 Runtime Semantics: IteratorBindingInitialization

With parameters iteratorRecord and environment.

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

ArrayBindingPattern : [ ]

1. Return NormalCompletion(empty).

ArrayBindingPattern : [ Elision ]

1. Return the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.

ArrayBindingPattern : [ Elision BindingRestElement ]

1. If Elision is present, then
   a. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
   b. ReturnIfAbrupt(status).
2. Return the result of performing IteratorBindingInitialization for BindingRestElement with iteratorRecord and environment as arguments.

ArrayBindingPattern : [ BindingElementList ]

1. Return the result of performing IteratorBindingInitialization for BindingElementList with iteratorRecord and environment as arguments.
ArrayBindingPattern : [ BindingElementList , ]

1. Return the result of performing IteratorBindingInitialization for BindingElementList with iteratorRecord and environment as arguments.

ArrayBindingPattern : [ BindingElementList , Elision ]

1. Let status be the result of performing IteratorBindingInitialization for BindingElementList with iteratorRecord and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.

ArrayBindingPattern : [ BindingElementList , Elision BindingRestElement ]

1. Let status be the result of performing IteratorBindingInitialization for BindingElementList with iteratorRecord and environment as arguments.
2. ReturnIfAbrupt(status).
3. If Elision is present, then
   a. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iteratorRecord as the argument.
   b. ReturnIfAbrupt(status).
4. Return the result of performing IteratorBindingInitialization for BindingRestElement with iteratorRecord and environment as arguments.

BindingElementList : BindingElisionElement

1. Return the result of performing IteratorBindingInitialization for BindingElisionElement with iteratorRecord and environment as arguments.

BindingElementList : BindingElementList , BindingElisionElement

1. Let status be the result of performing IteratorBindingInitialization for BindingElementList with iteratorRecord and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorBindingInitialization for BindingElisionElement using iteratorRecord and environment as arguments.

BindingElisionElement : BindingElement

1. Return the result of performing IteratorBindingInitialization of BindingElement with iteratorRecord and environment as the arguments.

BindingElisionElement : Elision BindingElement
1. Let \textit{status} be the result of performing \texttt{IteratorDestructuringAssignmentEvaluation} of \textit{Elision} with \textit{iteratorRecord} as the argument.
2. \texttt{ReturnIfAbrupt(status)}.
3. Return the result of performing \texttt{IteratorBindingInitialization} of \textit{BindingElement} with \textit{iteratorRecord} and \textit{environment} as the arguments.

\textit{BindingElement: SingleNameBinding}

1. Return the result of performing \texttt{IteratorBindingInitialization} for \textit{SingleNameBinding} with \textit{iteratorRecord} and \textit{environment} as the arguments.

\textit{SingleNameBinding: BindingIdentifier Initializer}

1. Let \textit{bindingId} be StringValue of \textit{BindingIdentifier}.
2. Let \textit{lhs} be ? \texttt{ResolveBinding(bindingId, environment)}.
3. If \texttt{iteratorRecord.}}[[\texttt{Done}]] is \texttt{false}, then
   a. Let \textit{next} be \texttt{IteratorStep(iteratorRecord.}[[\texttt{Iterator}]]).
   b. If \textit{next} is an abrupt completion, set \texttt{iteratorRecord.}[[\texttt{Done}]] to \texttt{true}.
   c. \texttt{ReturnIfAbrupt(next)}.
   d. If \textit{next} is \texttt{false}, set \texttt{iteratorRecord.}[[\texttt{Done}]] to \texttt{true}.
   e. Else,
      i. Let \textit{v} be \texttt{IteratorValue(next)}.
      ii. If \textit{v} is an abrupt completion, set \texttt{iteratorRecord.}[[\texttt{Done}]] to \texttt{true}.
      iii. \texttt{ReturnIfAbrupt(v)}.
4. If \texttt{iteratorRecord.}}[[\texttt{Done}]] is \texttt{true}, let \textit{v} be \texttt{undefined}.
5. If \texttt{Initializer} is present and \textit{v} is \texttt{undefined}, then
   a. Let \textit{defaultValue} be the result of evaluating \textit{Initializer}.
   b. Set \textit{v} to ? \texttt{GetValue(defaultValue)}.
   c. If \texttt{IsAnonymousFunctionDefinition(Initializer)} is \texttt{true}, then
      i. Let \textit{hasOwnProperty} be ? \texttt{HasOwnProperty(v, "name")}.
      ii. If \textit{hasOwnProperty} is \texttt{false}, perform \texttt{SetOwnProperty(v, bindingId)}.
6. If \textit{environment} is \texttt{undefined}, return ? \texttt{PutValue(lhs, v)}.
7. Return \texttt{InitializeReferencedBinding(lhs, v)}.

\textit{BindingElement: BindingPattern Initializer}

1. If \texttt{iteratorRecord.}[[\texttt{Done}]] is \texttt{false}, then
   a. Let \textit{next} be \texttt{IteratorStep(iteratorRecord.}[[\texttt{Iterator}]]).
   b. If \textit{next} is an abrupt completion, set \texttt{iteratorRecord.}[[\texttt{Done}]] to \texttt{true}.
   c. \texttt{ReturnIfAbrupt(next)}.
   d. If \textit{next} is \texttt{false}, set \texttt{iteratorRecord.}[[\texttt{Done}]] to \texttt{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.[[Iterator]])`.
      ii. If `next` is an abrupt completion, set `iteratorRecord.[[Done]]` to `true`.
      iii. `ReturnIfAbrupt(next)`.
   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. Let `status` be `CreateDataProperty(A, ! ToString(n), nextValue)`.
   g. Assert: `status` is `true`.
   h. Increment \( n \) by `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.[[Iterator]])`.
      ii. If `next` is an abrupt completion, set `iteratorRecord.[[Done]]` to `true`.
      iii. `ReturnIfAbrupt(next)`.
   b. If `iteratorRecord.[[Done]]` is `true`, then
i. Return the result of performing BindingInitialization of BindingPattern with \( A \) and 
\( \text{environment} \) as the arguments.
c. Let \( nextValue \) be 
\( \text{IteratorValue}(next) \).
d. If \( nextValue \) is an abrupt completion, set \( \text{iteratorRecord}.[[\text{Done}]] \) to \( \text{true} \).
e. ReturnIfAbrupt(\( nextValue \)).
f. Let \( status \) be 
\( \text{CreateDataProperty}(A, ! \text{ToString}(n), nextValue) \).
g. Assert: \( status \) is \( \text{true} \).
h. Increment \( n \) by 1.

### 13.3.3.7 Runtime Semantics: KeyedBindingInitialization

With parameters \( value, \text{environment} \), and \( \text{propertyName} \).

**NOTE**

When \( \text{undefined} \) is passed for \( \text{environment} \) 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**

1. Let \( v \) be \( \text{? GetV}(value, \text{propertyName}) \).
2. If \( \text{Initializer} \) is present and \( v \) is \( \text{undefined} \), then
   a. Let \( \text{defaultValue} \) be the result of evaluating \( \text{Initializer} \).
   b. Set \( v \) to \( \text{? GetValue}(\text{defaultValue}) \).
3. Return the result of performing BindingInitialization for BindingPattern passing \( v \) and \( \text{environment} \) as arguments.

**SingleNameBinding : BindingIdentifier Initializer**

1. Let \( \text{bindingId} \) be StringValue of BindingIdentifier.
2. Let \( \text{lhs} \) be \( \text{? ResolveBinding}(\text{bindingId}, \text{environment}) \).
3. Let \( v \) be \( \text{? GetV}(value, \text{propertyName}) \).
4. If \( \text{Initializer} \) is present and \( v \) is \( \text{undefined} \), then
   a. Let \( \text{defaultValue} \) be the result of evaluating \( \text{Initializer} \).
   b. Set \( v \) to \( \text{? GetValue}(\text{defaultValue}) \).
   c. If IsAnonymousFunctionDefinition(\( \text{Initializer} \)) is \( \text{true} \), then
      i. Let \( \text{hasNameProperty} \) be \( ? \text{HasOwnProperty}(v, \"name\") \).
      ii. If \( \text{hasNameProperty} \) is \( \text{false} \), perform SetFunctionName(\( v, \text{bindingId} \)).
5. If \( \text{environment} \) is \( \text{undefined} \), return \( \text{? PutValue}(\text{lhs}, v) \).
6. Return InitializeReferencedBinding(\( \text{lhs}, v \)).
13.4 Empty Statement

Syntax

```
EmptyStatement :
    ;
```

13.4.1 Runtime Semantics: Evaluation

```
EmptyStatement : ;
```

1. Return `NormalCompletion(empty)`.

13.5 Expression Statement

Syntax

```
ExpressionStatement[Yield, Await] :
    [lookahead \notin \{, 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`. An `ExpressionStatement` cannot start with the two token sequence `let [` because that would make it ambiguous with a `let LexicalDeclaration` whose first `LexicalBinding` was an `ArrayBindingPattern`.

13.5.1 Runtime Semantics: Evaluation

```
ExpressionStatement : Expression ;
```

1. Let `exprRef` be the result of evaluating `Expression`.
2. Return `? GetValue(exprRef)`.

13.6 The if Statement

Syntax
IfStatement[Yield, Await, Return] :
   Statement[?Yield, ?Await, ?Return]

Each else for which the choice of associated if is ambiguous shall be associated with the nearest possible if that would otherwise have no corresponding else.

13.6.1 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.
```

13.6.2 Static Semantics: ContainsDuplicateLabels

With parameter 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.

13.6.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter 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.

13.6.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.

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 « ».

13.6.5 Static Semantics: VarDeclaredNames

IfStatement : if ( Expression ) Statement else Statement

1. Let names be VarDeclaredNames of the first Statement.
2. Append to names the elements of the VarDeclaredNames of the second Statement.
3. Return names.

IfStatement : if ( Expression ) Statement

1. Return the VarDeclaredNames of Statement.

13.6.6 Static Semantics: VarScopedDeclarations

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.
13.6.7 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))`.

13.7 Iteration Statements

Syntax

```
IterationStatement[Yield, Await, Return] :
  for ( [lookahead ∉ { let [] }] Expression[~In, ?Yield, ?Await] opt ;
    Statement[?Yield, ?Await, ?Return]
  for ( var VariableDeclarationList[~In, ?Yield, ?Await] ;
    Statement[?Yield, ?Await, ?Return]
  for ( [lookahead ∉ { let [] }] LeftHandSideExpression[?Yield, ?Await] in
```
  Statement[?Yield, ?Await, ?Return]
for ( ForDeclaration[?Yield, ?Await] in Expression[+In, ?Yield, ?Await] )
  Statement[?Yield, ?Await, ?Return]
for ( [lookahead ≠ let] LeftHandSideExpression[?Yield, ?Await] of
  Statement[?Yield, ?Await, ?Return]
for ( ForDeclaration[?Yield, ?Await] of AssignmentExpression[+In, ?Yield, ?Await] )
  Statement[?Yield, ?Await, ?Return]

ForDeclaration[?Yield, ?Await] :
  LetOrConst ForBinding[?Yield, ?Await]

ForBinding[?Yield, ?Await] :
  BindingIdentifier[?Yield, ?Await]
  BindingPattern[?Yield, ?Await]

NOTE This section is extended by Annex B.3.6.

13.7.1 Semantics

13.7.1.1 Static Semantics: Early Errors

IterationStatement :
  do Statement while ( Expression );
  while ( Expression ) Statement
  for ( Expression opt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
  for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement
  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

• 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.
13.7.1.2 Runtime Semantics: LoopContinues ( completion, labelSet )

The abstract operation LoopContinues with arguments completion and labelSet is defined by the following steps:

1. If completion.[[Type]] is normal, return true.
2. If completion.[[Type]] is not continue, return false.
3. If completion.[[Target]] is empty, return true.
4. If completion.[[Target]] is an element of labelSet, return true.
5. Return false.

**NOTE** Within the Statement part of an IterationStatement a ContinueStatement may be used to begin a new iteration.

13.7.2 The do-while Statement

13.7.2.1 Static Semantics: ContainsDuplicateLabels

With parameter labelSet.

IterationStatement : do Statement while ( Expression ) ;

1. Return ContainsDuplicateLabels of Statement with argument labelSet.

13.7.2.2 Static Semantics: ContainsUndefinedBreakTarget

With parameter labelSet.

IterationStatement : do Statement while ( Expression ) ;

1. Return ContainsUndefinedBreakTarget of Statement with argument labelSet.

13.7.2.3 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.

IterationStatement : do Statement while ( Expression ) ;

1. Return ContainsUndefinedContinueTarget of Statement with arguments iterationSet and « ».

13.7.2.4 Static Semantics: VarDeclaredNames

IterationStatement : do Statement while ( Expression ) ;

1. Return the VarDeclaredNames of Statement.
13.7.2.5 Static Semantics: VarScoped Declarations
IterationStatement : do Statement while ( Expression ) ;

1. Return the VarScopedDeclarations of Statement.

13.7.2.6 Runtime Semantics: Labelled Evaluation

With parameter labelSet.

IterationStatement : 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).

13.7.3 The while Statement

13.7.3.1 Static Semantics: Contains Duplicate Labels

With parameter labelSet.

IterationStatement : while ( Expression ) Statement

1. Return ContainsDuplicateLabels of Statement with argument labelSet.

13.7.3.2 Static Semantics: Contains Undefined Break Target

With parameter labelSet.

IterationStatement : while ( Expression ) Statement

1. Return ContainsUndefinedBreakTarget of Statement with argument labelSet.

13.7.3.3 Static Semantics: Contains Undefined Continue Target

With parameters iterationSet and labelSet.
**IterationStatement**: `while ( Expression ) Statement`

1. Return `ContainsUndefinedContinueTarget` of `Statement` with arguments `iterationSet` and « ».

### 13.7.3.4 Static Semantics: VarDeclaredNames

**IterationStatement**: `while ( Expression ) Statement`

1. Return the `VarDeclaredNames` of `Statement`.

### 13.7.3.5 Static Semantics: VarScopedDeclarations

**IterationStatement**: `while ( Expression ) Statement`

1. Return the `VarScopedDeclarations` of `Statement`.

### 13.7.3.6 Runtime Semantics: LabelledEvaluation

With parameter `labelSet`.

**IterationStatement**: `while ( Expression ) Statement`

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

### 13.7.4 The for Statement

#### 13.7.4.1 Static Semantics: Early Errors

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

- It is a Syntax Error if any element of the `BoundNames` of `LexicalDeclaration` also occurs in the `VarDeclaredNames` of `Statement`.

#### 13.7.4.2 Static Semantics: ContainsDuplicateLabels

With parameter `labelSet`.
IterationStatement:
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
  for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement

1. Return ContainsDuplicateLabels of Statement with argument labelSet.

13.7.4.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter labelSet.

IterationStatement:
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
  for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement

1. Return ContainsUndefinedBreakTarget of Statement with argument labelSet.

13.7.4.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.

IterationStatement:
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
  for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement

1. Return ContainsUndefinedContinueTarget of Statement with arguments iterationSet and « ».

13.7.4.5 Static Semantics: VarDeclaredNames

IterationStatement: for ( Expression ; Expression ; Expression ) Statement

1. Return the VarDeclaredNames of Statement.

IterationStatement: for ( var VariableDeclarationList ; Expression ; Expression ) Statement

1. Let names be BoundNames of VariableDeclarationList.
2. Append to names the elements of the VarDeclaredNames of Statement.
3. Return names.

IterationStatement: for ( LexicalDeclaration Expression ; Expression ) Statement

1. Return the VarDeclaredNames of Statement.
13.7.4.6 Static Semantics: VarScopedDeclarations

IterationStatement : for ( Expression ; Expression ; Expression ) Statement

1. Return the VarScopedDeclarations of Statement.

IterationStatement : for ( var VariableDeclarationList ; Expression ; Expression ) Statement

1. Let declarations be VarScopedDeclarations of VariableDeclarationList.
2. Append to declarations the elements of the VarScopedDeclarations of Statement.
3. Return declarations.

IterationStatement : for ( LexicalDeclaration Expression ; Expression ) Statement

1. Return the VarScopedDeclarations of Statement.

13.7.4.7 Runtime Semantics: LabelledEvaluation

With parameter labelSet.

IterationStatement : for ( Expression ; Expression ; Expression ) 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).

IterationStatement : for ( var VariableDeclarationList ; Expression ; Expression ) Statement

1. Let varDcl be the result of evaluating VariableDeclarationList.
2. ReturnIfAbrupt(varDcl).
3. Return ? ForBodyEvaluation(the first Expression, the second Expression, Statement, « », labelSet).

IterationStatement : for ( LexicalDeclaration Expression ; Expression ) Statement

1. Let oldEnv be the running execution context’s LexicalEnvironment.
2. Let loopEnv be NewDeclarativeEnvironment(oldEnv).
3. Let loopEnvRec be loopEnv’s EnvironmentRecord.
4. Let isConst be the result of performing IsConstantDeclaration of LexicalDeclaration.
5. Let boundNames be the BoundNames of LexicalDeclaration.
6. For each element dn of boundNames, do
   a. If isConst is true, then
      i. Perform ! loopEnvRec.CreateImmutableBinding(dn, true).
   b. Else,
      i. Perform ! loopEnvRec.CreateMutableBinding(dn, false).
7. Set the running execution context's LexicalEnvironment to loopEnv.
8. Let forDcl be the result of evaluating LexicalDeclaration.
9. If forDcl is an abrupt completion, then
   a. Set the running execution context's LexicalEnvironment to oldEnv.
   b. Return Completion(forDcl).
10. If isConst is false, let perIterationLets be boundNames; otherwise let perIterationLets be « ».
11. Let bodyResult be ForBodyEvaluation(the first Expression, the second Expression, Statement, perIteration Lets, labelSet).
12. Set the running execution context's LexicalEnvironment to oldEnv.
13. Return Completion(bodyResult).

13.7.4.8 Runtime Semantics: ForBodyEvaluation( test, increment, stmt, perIterationBindings, labelSet )

The abstract operation ForBodyEvaluation with arguments test, increment, stmt, perIterationBindings, and labelSet is performed as follows:

1. Let V be undefined.
2. Perform ? CreatePerIterationEnvironment(perIterationBindings).
3. Repeat,
   a. If test is not [empty], then
      i. Let testRef be the result of evaluating test.
      ii. Let testValue be ? GetValue(testRef).
      iii. If ToBoolean(testValue) is false, return NormalCompletion(V).
   b. Let result be the result of evaluating stmt.
   c. If LoopContinues(result, labelSet) is false, return Completion(UpdateEmpty(result, V)).
   d. If result.([Value]) is not empty, set V to result.([Value]).
   e. Perform ? CreatePerIterationEnvironment(perIterationBindings).
   f. If increment is not [empty], then
      i. Let incRef be the result of evaluating increment.
      ii. Perform ? GetValue(incRef).

13.7.4.9 Runtime Semantics: CreatePerIterationEnvironment( perIterationBindings )

The abstract operation CreatePerIterationEnvironment with argument perIterationBindings is performed as follows:

1. If perIterationBindings has any elements, then
   a. Let lastIterationEnv be the running execution context's LexicalEnvironment.
   b. Let lastIterationEnvRec be lastIterationEnv's EnvironmentRecord.
   c. Let outer be lastIterationEnv's outer environment reference.
d. Assert: `outer` is not `null`.

e. Let `thisIterationEnv` be `NewDeclarativeEnvironment(outer)`.

f. Let `thisIterationEnvRec` be `thisIterationEnv`'s EnvironmentRecord.

g. For each element `bn` of `perIterationBindings`, do
   i. Perform `thisIterationEnvRec.CreateMutableBinding(bn, false)`.
   ii. Let `lastValue` be `? lastIterationEnvRec.GetBindingValue(bn, true)`.
   iii. Perform `thisIterationEnvRec.InitializeBinding(bn, lastValue)`.

h. Set the running execution context's LexicalEnvironment to `thisIterationEnv`.

2. Return `undefined`.

### 13.7.5 The for-in and for-of Statements

#### 13.7.5.1 Static Semantics: Early Errors

`IterationStatement`:

- `for ( LeftHandSideExpression in Expression ) Statement`
- `for ( LeftHandSideExpression of AssignmentExpression ) Statement`

- It is a Syntax Error if `LeftHandSideExpression` is either an `ObjectLiteral` or an `ArrayLiteral` and if the lexical token sequence matched by `LeftHandSideExpression` cannot be parsed with no tokens left over using `AssignmentPattern` as the goal symbol.

If `LeftHandSideExpression` is either an `ObjectLiteral` or an `ArrayLiteral` and if the lexical token sequence matched by `LeftHandSideExpression` can be parsed with no tokens left over using `AssignmentPattern` as the goal symbol then the following rules are not applied. Instead, the Early Error rules for `AssignmentPattern` are used.

- It is a Syntax Error if `IsValidSimpleAssignmentTarget of LeftHandSideExpression` is `false`.

- It is a Syntax Error if the `LeftHandSideExpression` is `CoverParenthesizedExpressionAndArrowParameterList : ( Expression )` and `Expression` derives a phrase that would produce a Syntax Error according to these rules if that phrase were substituted for `LeftHandSideExpression`. This rule is recursively applied.

**NOTE**

The last rule means that the other rules are applied even if parentheses surround `Expression`.

`IterationStatement`:

- `for ( ForDeclaration in Expression ) Statement`
- `for ( ForDeclaration of AssignmentExpression ) Statement`

- It is a Syntax Error if the BoundNames of `ForDeclaration` contains "let".
• It is a Syntax Error if any element of the BoundNames of \textit{ForDeclaration} also occurs in the VarDeclaredNames of \textit{Statement}.

• It is a Syntax Error if the BoundNames of \textit{ForDeclaration} contains any duplicate entries.

13.7.5.2 Static Semantics: BoundNames

\textit{ForDeclaration} : \textit{LetOrConst ForBinding}

1. Return the BoundNames of \textit{ForBinding}.

13.7.5.3 Static Semantics: ContainsDuplicateLabels

With parameter \textit{labelSet}.

\textit{IterationStatement} :

\begin{align*}
& \text{for ( LeftHandSideExpression in Expression ) Statement} \\
& \text{for ( var ForBinding in Expression ) Statement} \\
& \text{for ( ForDeclaration in Expression ) Statement} \\
& \text{for ( LeftHandSideExpression of AssignmentExpression ) Statement} \\
& \text{for ( var ForBinding of AssignmentExpression ) Statement} \\
& \text{for ( ForDeclaration of AssignmentExpression ) Statement}
\end{align*}

1. Return \text{ContainsDuplicateLabels} of \textit{Statement} with argument \textit{labelSet}.

\begin{itemize}
\item \textbf{NOTE} This section is extended by Annex B.3.6.
\end{itemize}

13.7.5.4 Static Semantics: ContainsUndefinedBreakTarget

With parameter \textit{labelSet}.

\textit{IterationStatement} :

\begin{align*}
& \text{for ( LeftHandSideExpression in Expression ) Statement} \\
& \text{for ( var ForBinding in Expression ) Statement} \\
& \text{for ( ForDeclaration in Expression ) Statement} \\
& \text{for ( LeftHandSideExpression of AssignmentExpression ) Statement} \\
& \text{for ( var ForBinding of AssignmentExpression ) Statement} \\
& \text{for ( ForDeclaration of AssignmentExpression ) Statement}
\end{align*}

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

\begin{itemize}
\item \textbf{NOTE} This section is extended by Annex B.3.6.
\end{itemize}

13.7.5.5 Static Semantics: ContainsUndefinedContinueTarget
With parameters $iterationSet$ and $labelSet$.

**IterationStatement**:

- `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`

1. Return `ContainsUndefinedContinueTarget` of $Statement$ with arguments $iterationSet$ and « ».

**NOTE**

This section is extended by Annex B.3.6.

### 13.7.5.6 Static Semantics: IsDestructuring

**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.

### 13.7.5.7 Static Semantics: VarDeclaredNames

**IterationStatement** : `for ( LeftHandSideExpression in Expression ) Statement`

1. Return the `VarDeclaredNames` of $Statement$.

**IterationStatement** : `for ( var ForBinding in Expression ) Statement`

1. Let `names` be the `BoundNames` of $ForBinding$.
2. Append to `names` the elements of the `VarDeclaredNames` of $Statement$.
3. Return `names`.

**IterationStatement** : `for ( ForDeclaration in Expression ) Statement`

1. Return the `VarDeclaredNames` of $Statement$.

**IterationStatement** : `for ( LeftHandSideExpression of AssignmentExpression ) Statement`
1. Return the VarDeclaredNames of Statement.

\[\text{IterationStatement} : \text{for ( var ForBinding of AssignmentExpression ) Statement}\]

1. Let names be the BoundNames of ForBinding.
2. Append to names the elements of the VarDeclaredNames of Statement.
3. Return names.

\[\text{IterationStatement} : \text{for ( ForDeclaration of AssignmentExpression ) Statement}\]

1. Return the VarDeclaredNames of Statement.

\begin{Verbatim}
\textbf{NOTE} \quad \text{This section is extended by Annex B.3.6.}
\end{Verbatim}

\subsection*{13.7.5.8 Static Semantics: Var Scoped Declarations}
\[\text{IterationStatement} : \text{for ( LeftHandSideExpression in Expression ) Statement}\]

1. Return the VarScopedDeclarations of Statement.

\[\text{IterationStatement} : \text{for ( var ForBinding in Expression ) Statement}\]

1. Let declarations be a List containing ForBinding.
2. Append to declarations the elements of the VarScopedDeclarations of Statement.
3. Return declarations.

\[\text{IterationStatement} : \text{for ( ForDeclaration in Expression ) Statement}\]

1. Return the VarScopedDeclarations of Statement.

\[\text{IterationStatement} : \text{for ( LeftHandSideExpression of AssignmentExpression ) Statement}\]

1. Return the VarScopedDeclarations of Statement.

\[\text{IterationStatement} : \text{for ( var ForBinding of AssignmentExpression ) Statement}\]

1. Let declarations be a List containing ForBinding.
2. Append to declarations the elements of the VarScopedDeclarations of Statement.
3. Return declarations.

\[\text{IterationStatement} : \text{for ( ForDeclaration of AssignmentExpression ) Statement}\]

1. Return the VarScopedDeclarations of Statement.

\begin{Verbatim}
\textbf{NOTE} \quad \text{This section is extended by Annex B.3.6.}
\end{Verbatim}

\subsection*{13.7.5.9 Runtime Semantics: Binding Initialization}
With parameters `value` and `environment`.

**NOTE** `undefined` is passed for `environment` to indicate that a `PutValue` operation should be used to assign the initialization value. This is the case for `var` statements and the formal parameter lists of some non-strict functions (see 9.2.12). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

*ForDeclaration* : `LetOrConst ForBinding`

1. Return the result of performing `BindingInitialization` for `ForBinding` passing `value` and `environment` as the arguments.

### 13.7.5.10 Runtime Semantics: `BindingInstantiation`

With parameter `environment`.

*ForDeclaration* : `LetOrConst ForBinding`

1. Let `envRec` be `environment`'s `EnvironmentRecord`.
2. Assert: `envRec` is a declarative `EnvironmentRecord`.
3. For each element `name` of the `BoundNames` of `ForBinding`, do
   a. If `IsConstantDeclaration` of `LetOrConst` is `true`, then
      i. Perform `! envRec.CreateImmutableBinding(name, true)`.
   b. Else,
      i. Perform `! envRec.CreateMutableBinding(name, false)`.

### 13.7.5.11 Runtime Semantics: `LabelledEvaluation`

With parameter `labelSet`.

*IterationStatement* : `for` ( `LeftHandSideExpression in Expression` ) `Statement`

1. Let `keyResult` be `ForIn/OfHeadEvaluation`(`« »`, `Expression`, `enumerate`).
2. Return `ForIn/OfBodyEvaluation`(`LeftHandSideExpression`, `Statement`, `keyResult`, `enumerate`, `assignment`, `labelSet`).

*IterationStatement* : `for` ( `var ForBinding in Expression` ) `Statement`

1. Let `keyResult` be `ForIn/OfHeadEvaluation`(`« »`, `Expression`, `enumerate`).
2. Return `ForIn/OfBodyEvaluation`(`ForBinding`, `Statement`, `keyResult`, `enumerate`, `varBinding`, `labelSet`).

*IterationStatement* : `for` ( `ForDeclaration in Expression` ) `Statement`
1. Let `keyResult` be the result of performing `? ForIn/OfHeadEvaluation(BoundNames of ForDeclaration, Expression, enumerate)`.

2. Return `? ForIn/OfBodyEvaluation(ForDeclaration, Statement, keyResult, enumerate, lexicalBinding, labelSet)`.

**IterationStatement** : `for ( LeftHandSideExpression of AssignmentExpression ) Statement`

1. Let `keyResult` be the result of performing `? ForIn/OfHeadEvaluation(« », AssignmentExpression, iterate)`.

2. Return `? ForIn/OfBodyEvaluation(LeftHandSideExpression, Statement, keyResult, iterate, assignment, labelSet)`.

**IterationStatement** : `for ( var ForBinding of AssignmentExpression ) Statement`

1. Let `keyResult` be the result of performing `? ForIn/OfHeadEvaluation(« », AssignmentExpression, iterate)`.

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

**IterationStatement** : `for ( ForDeclaration of AssignmentExpression ) Statement`

1. Let `keyResult` be the result of performing `? ForIn/OfHeadEvaluation(BoundNames of ForDeclaration, AssignmentExpression, iterate)`.

2. Return `? ForIn/OfBodyEvaluation(ForDeclaration, Statement, keyResult, iterate, lexicalBinding, labelSet)`.

---

**NOTE**
This section is extended by Annex B.3.6.

---

**13.7.5.12 Runtime Semantics: ForIn/OfHeadEvaluation (TDZnames, expr, iterationKind)**

The abstract operation ForIn/OfHeadEvaluation is called with arguments `TDZnames`, `expr`, and `iterationKind`. The value of `iterationKind` is either `enumerate` or `iterate`.

1. Let `oldEnv` be the running execution context’s LexicalEnvironment.

2. If `TDZnames` is not an empty List, then
   a. Assert: `TDZnames` has no duplicate entries.
   b. Let `TDZ` be `NewDeclarativeEnvironment(oldEnv)`.
   c. Let `TDZEnvRec` be `TDZ`’s EnvironmentRecord.
   d. For each string `name` in `TDZnames`, do
      i. Perform `! TDZEnvRec.CreateMutableBinding(name, false)`.
   e. Set the running execution context’s LexicalEnvironment to `TDZ`.

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. Return ? **EnumerateObjectProperties**(*obj*).

7. Else,
   a. Assert: *iterationKind* is *iterate*.
   b. Return ? **GetIterator**(*exprValue*).

13.7.5.13 Runtime Semantics: ForIn/OfBodyEvaluation ( *lhs*, *stmt*, *iterator*, *iterationKind*, *lhsKind*, *labelSet*)

The abstract operation ForIn/OfBodyEvaluation is called with arguments *lhs*, *stmt*, *iterator*, *iterationKind*, *lhsKind*, and *labelSet*. The value of *iterationKind* is either *enumerate* or *iterate*. The value of *lhsKind* is either assignment, varBinding or lexicalBinding.

1. Let *oldEnv* be the running execution context’s LexicalEnvironment.
2. Let *V* be *undefined*.
3. Let *destructuring* be IsDestructuring of *lhs*.
4. If *destructuring* is *true* and if *lhsKind* is assignment, then
   a. Assert: *lhs* is a *LeftHandSideExpression*.
   b. Let *assignmentPattern* be the parse of the source text corresponding to *lhs* using *AssignmentPattern* as the goal symbol.
5. Repeat,
   a. Let *nextResult* be ? **IteratorStep**(*iterator*).
   b. If *nextResult* is *false*, return *NormalCompletion*(*V*).
   c. Let *nextValue* be ? **IteratorValue**(*nextResult*).
   d. 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.)
   e. Else,
      i. Assert: *lhsKind* is lexicalBinding.
      ii. Assert: *lhs* is a *ForDeclaration*.
      iii. Let *iterationEnv* be **NewDeclarativeEnvironment**(*oldEnv*).
      iv. Perform BindingInstantiation 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*).
   f. 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 \texttt{InitializeReferencedBinding(}\( lhsRef, nextValue \)).
iii. Else,
   1. Let \( status \) be \texttt{PutValue(}\( lhsRef, nextValue \)).
g. Else,
i. If \( lhsKind \) is assignment, then
   1. Let \( status \) be the result of performing DestructuringAssignmentEvaluation of 
      \texttt{assignmentPattern} using \( nextValue \) as the argument.
ii. Else if \( lhsKind \) is varBinding, then
   1. Assert: \( lhs \) is a \texttt{ForBinding}.
   2. Let \( status \) be the result of performing BindingInitialization for \( lhs \) passing \( nextValue \) 
      and \texttt{undefined} as the arguments.
iii. Else,
   1. Assert: \( lhsKind \) is lexicalBinding.
   2. Assert: \( lhs \) is a \texttt{ForDeclaration}.
   3. Let \( status \) be the result of performing BindingInitialization for \( lhs \) passing \( nextValue \) 
      and \texttt{iterationEnv} as arguments.
h. If \( status \) is an abrupt completion, then
   i. Set the running execution context’s LexicalEnvironment to \texttt{oldEnv}.
   ii. If \( iterationKind \) is enumerate, then
      1. Return \( status \).
   iii. Else,
      1. Assert: \( iterationKind \) is iterate.
      2. Return \( ?\texttt{IteratorClose(iterator, status)} \).
i. Let \( result \) be the result of evaluating \( stmt \).
j. Set the running execution context’s LexicalEnvironment to \texttt{oldEnv}.
k. If \texttt{LoopContinues(}\( result, labelSet \)) is \texttt{false}, then
   i. If \( iterationKind \) is enumerate, then
      1. Return \texttt{Completion(UpdateEmpty(result, V))}.
   ii. Else,
      1. Assert: \( iterationKind \) is iterate.
      2. Return \( ?\texttt{IteratorClose(iterator, UpdateEmpty(result, V))} \).
l. If \( result[[Value]] \) is not empty, set \( V \) to \( result[[Value]] \).

13.7.5.14 Runtime Semantics: Evaluation

\texttt{ForBinding : BindingIdentifier} 

1. Let \texttt{bindingId} be StringValue of \texttt{BindingIdentifier}.
2. Return ? `ResolveBinding(bindingId)`.

### 13.7.5.15 `EnumerateObjectProperties (O)`

When the abstract operation `EnumerateObjectProperties` is called with argument `O`, the following steps are taken:

1. Assert: `Type(O)` is `Object`.
2. Return an Iterator object (25.1.2) 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.

```
NOTE

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.has(key)) {
            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;
}

13.8 The continue Statement

Syntax

`ContinueStatement[Yield, Await] : continue ;

13.8.1 Static Semantics: Early Errors

`ContinueStatement : continue ;
ContinueStatement : continue LabelIdentifier ;`

- It is a Syntax Error if this `ContinueStatement` is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement`.

13.8.2 Static Semantics: ContainsUndefinedContinueTarget

With parameters `iterationSet` and `labelSet`.

`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`.

13.8.3 Runtime Semantics: Evaluation

`ContinueStatement : continue ;`
1. Return `Completion`{[[Type]]: continue, [[Value]]: empty, [[Target]]: empty}.

`ContinueStatement` : `continue LabelIdentifier` ;

1. Let `label` be the `StringValue` of `LabelIdentifier`.
2. Return `Completion`{[[Type]]: continue, [[Value]]: empty, [[Target]]: `label`}.

### 13.9 The break Statement

#### Syntax

```plaintext
BreakStatement[Yield, Await] :
    break ;
```

#### 13.9.1 Static Semantics: Early Errors

`BreakStatement` : `break` ;

- It is a Syntax Error if this `BreakStatement` is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement` or a `SwitchStatement`.

#### 13.9.2 Static Semantics: `ContainsUndefinedBreakTarget`

With parameter `labelSet`.

`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`.

#### 13.9.3 Runtime Semantics: Evaluation

`BreakStatement` : `break` ;

1. Return `Completion`{[[Type]]: break, [[Value]]: empty, [[Target]]: empty}.
1. Let label be the StringValue of LabelIdentifier.
2. Return Completion{[[Type]]: break, [[Value]]: empty, [[Target]]: label}.

13.10 The return Statement

Syntax

```
ReturnStatement[Yield, Await] :
  return ;
```

NOTE A return statement causes a function to cease execution and return a value to the caller. If Expression is omitted, the return value is undefined. Otherwise, the return value is the value of Expression.

13.10.1 Runtime Semantics: Evaluation

```
ReturnStatement : return ;
```

1. Return Completion{[[Type]]: return, [[Value]]: undefined, [[Target]]: empty}.

```
ReturnStatement : return Expression ;
```

1. Let exprRef be the result of evaluating Expression.
2. Let exprValue be ?GetValue(exprRef).
3. Return Completion{[[Type]]: return, [[Value]]: exprValue, [[Target]]: empty}.

13.11 The with Statement

Syntax

```
WithStatement[Yield, Await, Return] :
```

NOTE The with statement adds an object Environment Record for a computed object to the lexical environment of the running execution context. It then executes a statement using this augmented lexical environment. Finally, it restores the original lexical environment.

13.11.1 Static Semantics: Early Errors

```
WithStatement : with ( Expression ) Statement
```
It is a Syntax Error if the code that matches this production is contained in strict mode code.

It is a Syntax Error if IsLabelledFunction(Statement) is true.

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

### 13.11.2 Static Semantics: ContainsDuplicateLabels

With parameter `labelSet`.

```plaintext
WithStatement : with ( Expression ) Statement

1. Return ContainsDuplicateLabels of Statement with argument labelSet.
```

### 13.11.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter `labelSet`.

```plaintext
WithStatement : with ( Expression ) Statement

1. Return ContainsUndefinedBreakTarget of Statement with argument labelSet.
```

### 13.11.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters `iterationSet` and `labelSet`.

```plaintext
WithStatement : with ( Expression ) Statement

1. Return ContainsUndefinedContinueTarget of Statement with arguments iterationSet and « ».
```

### 13.11.5 Static Semantics: VarDeclaredNames

```plaintext
WithStatement : with ( Expression ) Statement

1. Return the VarDeclaredNames of Statement.
```

### 13.11.6 Static Semantics: VarScopedDeclarations

```plaintext
WithStatement : with ( Expression ) Statement

1. Return the VarScopedDeclarations of Statement.
```

### 13.11.7 Runtime Semantics: Evaluation
WithStatement : with ( Expression ) Statement

1. Let val be the result of evaluating Expression.
2. Let obj be ?ToObject(? GetValue(val)).
3. Let oldEnv be the running execution context's LexicalEnvironment.
4. Let newEnv be NewObjectEnvironment(obj, oldEnv).
5. Set the withEnvironment flag of newEnv's EnvironmentRecord to true.
6. Set the running execution context's LexicalEnvironment to newEnv.
7. Let C be the result of evaluating Statement.
8. Set the running execution context's LexicalEnvironment to oldEnv.
9. Return Completion(UpdateEmpty(C, undefined)).

NOTE No matter how control leaves the embedded Statement, whether normally or by some form of abrupt completion or exception, the LexicalEnvironment is always restored to its former state.

13.12 The switch Statement

Syntax

SwitchStatement[Yield, Await, Return] :

CaseBlock[Yield, Await, Return] :
   { CaseClauses[?Yield, ?Await, ?Return] opt }
   CaseClauses[?Yield, ?Await, ?Return] opt

CaseClauses[Yield, Await, Return] :
   CaseClause[?Yield, ?Await, ?Return]

CaseClause[Yield, Await, Return] :

DefaultClause[Yield, Await, Return] :

13.12.1 Static Semantics: Early Errors

SwitchStatement : switch ( Expression ) CaseBlock
• It is a Syntax Error if the LexicallyDeclaredNames of `CaseBlock` contains any duplicate entries.
• It is a Syntax Error if any element of the LexicallyDeclaredNames of `CaseBlock` also occurs in the VarDeclaredNames of `CaseBlock`.

### 13.12.2 Static Semantics: ContainsDuplicateLabels

With parameter `labelSet`.

```
SwitchStatement : switch ( Expression ) CaseBlock
```

1. Return `ContainsDuplicateLabels` of `CaseBlock` with argument `labelSet`.

```
CaseBlock : {  }
```

1. Return `false`.

```
CaseBlock : {  CaseClauses DefaultClause CaseClauses  }
```

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
```

1. If the `StatementList` is present, return `ContainsDuplicateLabels` of `StatementList` with argument `labelSet`.
2. Return `false`.

```
DefaultClause : default : StatementList
```

1. If the `StatementList` is present, return `ContainsDuplicateLabels` of `StatementList` with argument `labelSet`.
2. Return `false`. 
13.12.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter labelSet.

SwitchStatement : switch ( Expression ) CaseBlock

1. Return ContainsUndefinedBreakTarget of CaseBlock with argument labelSet.

CaseBlock : { }

1. Return false.

CaseBlock : { CaseClauses DefaultClause CaseClauses }

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

CaseClauses : CaseClauses CaseClause

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

CaseClause : case Expression : StatementList

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

DefaultClause : default : StatementList

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

13.12.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.
SwitchStatement : \texttt{switch ( Expression ) CaseBlock}

1. Return 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{CaseBlock} with arguments \texttt{iterationSet} and « ».

CaseBlock : \{ \}

1. Return \texttt{false}.

CaseBlock : \{ CaseClauses DefaultClause CaseClauses \}

1. If the first \texttt{CaseClauses} is present, then
   a. Let \texttt{hasUndefinedLabels} be 
      \texttt{ContainsUndefinedContinueTarget} of the first \texttt{CaseClauses} with
         arguments \texttt{iterationSet} and « ».
   b. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
2. Let \texttt{hasUndefinedLabels} be 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{DefaultClause} with arguments
      \texttt{iterationSet} and « ».
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{ContainsUndefinedContinueTarget} of the second \texttt{CaseClauses} with arguments \texttt{iterationSet} and « ».

CaseClauses : CaseClauses CaseClause

1. Let \texttt{hasUndefinedLabels} be 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{CaseClauses} with arguments
      \texttt{iterationSet} and « ».
2. If \texttt{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
3. Return 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{CaseClause} with arguments \texttt{iterationSet} and « ».

CaseClause : \texttt{case} Expression : StatementList

1. If the \texttt{StatementList} is present, return 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{StatementList} with
      arguments \texttt{iterationSet} and « ».
2. Return \texttt{false}.

DefaultClause : \texttt{default} : StatementList

1. If the \texttt{StatementList} is present, return 
   \texttt{ContainsUndefinedContinueTarget} of \texttt{StatementList} with
      arguments \texttt{iterationSet} and « ».
2. Return \texttt{false}.

13.12.5 Static Semantics: LexicallyDeclaredNames

CaseBlock : \{ \}

1. Return a new empty \texttt{List}.
CaseBlock : { CaseClauses DefaultClause CaseClauses }

1. If the first CaseClauses is present, let names be the LexicallyDeclaredNames of the first CaseClauses.
2. Else, let names be a new empty List.
3. Append to names the elements of the LexicallyDeclaredNames of the DefaultClause.
4. If the second CaseClauses is not present, return names.
5. Return the result of appending to names the elements of the LexicallyDeclaredNames of the second CaseClauses.

CaseClauses : CaseClauses CaseClause

1. Let names be LexicallyDeclaredNames of CaseClauses.
2. Append to names the elements of the LexicallyDeclaredNames of CaseClause.
3. Return names.

CaseClause : case Expression : StatementList

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

DefaultClause : default : StatementList

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

13.12.6 Static Semantics: LexicallyScopedDeclarations

CaseBlock : { }

1. Return a new empty List.

CaseBlock : { CaseClauses DefaultClause CaseClauses }

1. If the first CaseClauses is present, let declarations be the LexicallyScopedDeclarations of the first CaseClauses.
2. Else, let declarations be a new empty List.
3. Append to declarations the elements of the LexicallyScopedDeclarations of the DefaultClause.
4. If the second CaseClauses is not present, return declarations.
5. 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*: \texttt{case Expression : StatementList}

\begin{enumerate}
\item If the *StatementList* is present, return the LexicallyScopedDeclarations of *StatementList*.
\item Return a new empty List.
\end{enumerate}

*DefaultClause*: \texttt{default : StatementList}

\begin{enumerate}
\item If the *StatementList* is present, return the LexicallyScopedDeclarations of *StatementList*.
\item Return a new empty List.
\end{enumerate}

### 13.12.7 Static Semantics: VarDeclaredNames

*SwitchStatement*: \texttt{switch ( Expression ) CaseBlock}

\begin{enumerate}
\item Return the VarDeclaredNames of *CaseBlock*.
\end{enumerate}

*CaseBlock*: \{\}

\begin{enumerate}
\item Return a new empty List.
\end{enumerate}

*CaseBlock*: \{ CaseClauses DefaultClause CaseClauses \}

\begin{enumerate}
\item If the first *CaseClauses* is present, let \texttt{names} be the VarDeclaredNames of the first *CaseClauses*.
\item Else, let \texttt{names} be a new empty List.
\item Append to \texttt{names} the elements of the VarDeclaredNames of the *DefaultClause*.
\item If the second *CaseClauses* is not present, return \texttt{names}.
\item Return the result of appending to \texttt{names} the elements of the VarDeclaredNames of the second *CaseClauses*.
\end{enumerate}

*CaseClauses*: CaseClauses CaseClause

\begin{enumerate}
\item Let \texttt{names} be VarDeclaredNames of *CaseClauses*.
\item Append to \texttt{names} the elements of the VarDeclaredNames of *CaseClause*.
\item Return \texttt{names}.
\end{enumerate}

*CaseClause*: \texttt{case Expression : StatementList}

\begin{enumerate}
\item If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
\item Return a new empty List.
\end{enumerate}

*DefaultClause*: \texttt{default : StatementList}

\begin{enumerate}
\item If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
\item Return a new empty List.
\end{enumerate}
13.12.8 Static Semantics: VarScopedDeclarations

SwitchStatement : \texttt{switch} ( Expression ) CaseBlock

1. Return the VarScopedDeclarations of \texttt{CaseBlock}.

\texttt{CaseBlock} : \{ \}

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

\texttt{CaseBlock} : \{ \texttt{CaseClauses} DefaultClause CaseClauses \}

1. If the first \texttt{CaseClauses} is present, let \texttt{declarations} be the VarScopedDeclarations of the first \texttt{CaseClauses}.
2. Else, let \texttt{declarations} be a new empty \texttt{List}.
3. Append to \texttt{declarations} the elements of the VarScopedDeclarations of the \texttt{DefaultClause}.
4. If the second \texttt{CaseClauses} is not present, return \texttt{declarations}.
5. Return the result of appending to \texttt{declarations} the elements of the VarScopedDeclarations of the second \texttt{CaseClauses}.

\texttt{CaseClauses} : \texttt{CaseClauses} CaseClause

1. Let \texttt{declarations} be VarScopedDeclarations of \texttt{CaseClauses}.
2. Append to \texttt{declarations} the elements of the VarScopedDeclarations of \texttt{CaseClause}.
3. Return \texttt{declarations}.

\texttt{CaseClause} : \texttt{case} Expression : StatementList

1. If the \texttt{StatementList} is present, return the VarScopedDeclarations of \texttt{StatementList}.
2. Return a new empty \texttt{List}.

\texttt{DefaultClause} : \texttt{default} : StatementList

1. If the \texttt{StatementList} is present, return the VarScopedDeclarations of \texttt{StatementList}.
2. Return a new empty \texttt{List}.

13.12.9 Runtime Semantics: CaseBlockEvaluation

With parameter \texttt{input}.

\texttt{CaseBlock} : \{ \}

1. Return \texttt{NormalCompletion(undefined)}.

\texttt{CaseBlock} : \{ \texttt{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 $C$ in $A$, do
   a. If $found$ is **false**, then
      i. Let clauseSelector be the result of CaseSelectorEvaluation of $C$.
      ii. ReturnIfAbrupt(clauseSelector).
      iii. Set $found$ to the result of performing Strict Equality Comparison $input === clauseSelector$.
   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** **DefaultClause** **CaseClauses** }

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 $C$ in $A$, do
   a. If $found$ is **false**, then
      i. Let clauseSelector be the result of CaseSelectorEvaluation of $C$.
      ii. ReturnIfAbrupt(clauseSelector).
      iii. Set $found$ to the result of performing Strict Equality Comparison $input === clauseSelector$.
   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 $C$ in $B$, do
      i. If $foundInB$ is **false**, then
         1. Let clauseSelector be the result of CaseSelectorEvaluation of $C$.
         2. ReturnIfAbrupt(clauseSelector).
3. Set `foundInB` to the result of performing `Strict Equality Comparison input == clauseSelector`.
   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. For each `CaseClause C` in `B` (NOTE: this is another complete iteration of the second `CaseClauses`), do
   a. Let `R` be the result of evaluating `CaseClause C`.
   b. If `R.[[Value]]` is not empty, set `V` to `R.[[Value]]`.
   c. If `R` is an abrupt completion, return `Completion(UpdateEmpty(R, V))`.
15. Return `NormalCompletion(V)`.

### 13.12.10 Runtime Semantics: `CaseSelectorEvaluation`

**CaseClause : case Expression : StatementList**

1. Let `exprRef` be the result of evaluating `Expression`.
2. Return `? getValue(exprRef)`.

**NOTE** CaseSelectorEvaluation does not execute the associated `StatementList`. It simply evaluates the `Expression` and returns the value, which the `CaseBlock` algorithm uses to determine which `StatementList` to start executing.

### 13.12.11 Runtime Semantics: `Evaluation`

**SwitchStatement : switch ( Expression ) CaseBlock**

1. Let `exprRef` be the result of evaluating `Expression`.
2. Let `switchValue` be `? getValue(exprRef)`.
3. Let `oldEnv` be the running execution context's LexicalEnvironment.
4. Let `blockEnv` be `NewDeclarativeEnvironment(oldEnv)`.
5. Perform `BlockDeclarationInstantiation(CaseBlock, blockEnv)`.
6. Set the running execution context's LexicalEnvironment to `blockEnv`.
7. Let `R` be the result of performing `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 : case Expression :

1. Return NormalCompletion(empty).

CaseClause : case Expression : StatementList

1. Return the result of evaluating StatementList.

DefaultClause : default :

1. Return NormalCompletion(empty).

DefaultClause : default : StatementList

1. Return the result of evaluating StatementList.

13.13 Labelled Statements

Syntax

LabelledStatement[Yield, Await, Return] :

LabelledItem[Yield, Await, Return] :
Statement[?Yield, ?Await, ?Return]
FunctionDeclaration[?Yield, ?Await, ~Default]

NOTE

A Statement may be prefixed by a label. Labelled statements are only used in conjunction with labelled break and continue statements. ECMAScript has no goto statement. A Statement can be part of a LabelledStatement, which itself can be part of a LabelledStatement, and so on. The labels introduced this way are collectively referred to as the “current label set” when describing the semantics of individual statements.

13.13.1 Static Semantics: Early Errors

LabelledItem : FunctionDeclaration

- It is a Syntax Error if any source text matches this rule.

NOTE

An alternative definition for this rule is provided in B.3.2.

13.13.2 Static Semantics: ContainsDuplicateLabels
With parameter \( \textit{labelSet} \).

\[
\textit{LabelledStatement} : \textit{LabelIdentifier} : \textit{LabelledItem}
\]

1. Let \( \textit{label} \) be the \textit{StringValue} of \( \textit{LabelIdentifier} \).
2. If \( \textit{label} \) is an element of \( \textit{labelSet} \), return \textit{true}.
3. Let \( \textit{newLabelSet} \) be a copy of \( \textit{labelSet} \) with \( \textit{label} \) appended.
4. Return \textit{ContainsDuplicateLabels} of \( \textit{LabelledItem} \) with argument \( \textit{newLabelSet} \).

\[
\textit{LabelledItem} : \textit{FunctionDeclaration}
\]

1. Return \textit{false}.

### 13.13.3 Static Semantics: \textit{ContainsUndefinedBreakTarget}

With parameter \( \textit{labelSet} \).

\[
\textit{LabelledStatement} : \textit{LabelIdentifier} : \textit{LabelledItem}
\]

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

\[
\textit{LabelledItem} : \textit{FunctionDeclaration}
\]

1. Return \textit{false}.

### 13.13.4 Static Semantics: \textit{ContainsUndefinedContinueTarget}

With parameters \textit{iterationSet} and \( \textit{labelSet} \).

\[
\textit{LabelledStatement} : \textit{LabelIdentifier} : \textit{LabelledItem}
\]

1. Let \( \textit{label} \) be the \textit{StringValue} of \( \textit{LabelIdentifier} \).
2. Let \( \textit{newLabelSet} \) be a copy of \( \textit{labelSet} \) with \( \textit{label} \) appended.
3. Return \textit{ContainsUndefinedContinueTarget} of \( \textit{LabelledItem} \) with arguments \textit{iterationSet} and \( \textit{newLabelSet} \).

\[
\textit{LabelledItem} : \textit{FunctionDeclaration}
\]

1. Return \textit{false}.

### 13.13.5 Static Semantics: \textit{IsLabelledFunction (stmt )}

The abstract operation IsLabelledFunction with argument \textit{stmt} performs the following steps:

1. If \textit{stmt} is not a \textit{LabelledStatement}, return \textbf{false}.
2. Let \textit{item} be the \textit{LabelledItem} of \textit{stmt}.
3. If \textit{item} is \textit{LabelledItem : FunctionDeclaration}, return \textbf{true}.
4. Let \textit{subStmt} be the \textit{Statement} of \textit{item}.
5. Return \textbf{IsLabelledFunction}(\textit{subStmt}).

**13.13.6 Static Semantics: LexicallyDeclaredNames**

\textit{LabelledStatement : LabelId : LabelledItem}

1. Return the LexicallyDeclaredNames of \textit{LabelledItem}.

**13.13.7 Static Semantics: LexicallyScopedDeclarations**

\textit{LabelledStatement : LabelId : LabelledItem}

1. Return the LexicallyScopedDeclarations of \textit{LabelledItem}.

**13.13.8 Static Semantics: TopLevelLexicallyDeclaredNames**

\textit{LabelledStatement : LabelId : LabelledItem}

1. Return a new empty list.

**13.13.9 Static Semantics: TopLevelLexicallyScopedDeclarations**

\textit{LabelledStatement : LabelId : LabelledItem}

1. Return a new empty list.
13.13.10 Static Semantics: TopLevelVarDeclaredNames
\[\text{LabelledStatement : LabelIdentifier : LabelledItem}\]

1. Return the TopLevelVarDeclaredNames of \textit{LabelledItem}.

\[\text{LabelledItem : Statement}\]

1. If \textit{Statement} is \textit{Statement} : \textit{LabelledStatement} , return TopLevelVarDeclaredNames of \textit{Statement}.
2. Return VarDeclaredNames of \textit{Statement}.

\[\text{LabelledItem : FunctionDeclaration}\]

1. Return BoundNames of \textit{FunctionDeclaration}.

13.13.11 Static Semantics: TopLevelVarScopedDeclarations
\[\text{LabelledStatement : LabelIdentifier : LabelledItem}\]

1. Return the TopLevelVarScopedDeclarations of \textit{LabelledItem}.

\[\text{LabelledItem : Statement}\]

1. If \textit{Statement} is \textit{Statement} : \textit{LabelledStatement} , return TopLevelVarScopedDeclarations of \textit{Statement}.
2. Return VarScopedDeclarations of \textit{Statement}.

\[\text{LabelledItem : FunctionDeclaration}\]

1. Return a new \textit{List} containing \textit{FunctionDeclaration}.

13.13.12 Static Semantics: VarDeclaredNames
\[\text{LabelledStatement : LabelIdentifier : LabelledItem}\]

1. Return the VarDeclaredNames of \textit{LabelledItem}.

\[\text{LabelledItem : FunctionDeclaration}\]

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

13.13.13 Static Semantics: VarScopedDeclarations
\[\text{LabelledStatement : LabelIdentifier : LabelledItem}\]

1. Return the VarScopedDeclarations of \textit{LabelledItem}.

\[\text{LabelledItem : FunctionDeclaration}\]
1. Return a new empty List.

### 13.13.14 Runtime Semantics: LabelledEvaluation

*With parameter labelSet.*

\[\text{LabelledStatement} : \text{LabelIdentifier} : \text{LabelledItem}\\
\]

1. Let \( label \) be the StringValue of \( \text{LabelIdentifier} \).
2. Append \( label \) as an element of \( \text{labelSet} \).
3. Let \( \text{stmtResult} \) be LabelledEvaluation of \( \text{LabelledItem} \) with argument \( \text{labelSet} \).
4. If \( \text{stmtResult}.[[\text{Type}]] \) is \text{break} and \( \text{SameValue}(\text{stmtResult}.[[\text{Target}]], \text{label}) \) is \text{true}, then
   a. Set \( \text{stmtResult} \) to \text{NormalCompletion}(\text{stmtResult}.[[\text{Value}]]).
5. Return \text{Completion}(\text{stmtResult}).

### 13.13.15 Runtime Semantics: Evaluation

*LabelledStatement : LabelIdentifier : LabelledItem*

1. Let \( \text{newLabelSet} \) be a new empty List.
2. Return LabelledEvaluation of this \( \text{LabelledStatement} \) with argument \( \text{newLabelSet} \).

### 13.14 The throw Statement

**Syntax**

\[
\text{ThrowStatement}[^{Yield, \text{Await}}]:\\
\quad \text{throw} \ [\text{no LineTerminator here}] \ \text{Expression}[^{+\text{In}, \text{?Yield, \text{?Await}}}] ;
\]

### 13.14.1 Runtime Semantics: Evaluation
ThrowStatement: `throw` Expression

1. Let `exprRef` be the result of evaluating `Expression`.
2. Let `exprValue` be ? `GetValue(exprRef)`.
3. Return `Completion`{[[Type]]: `throw`, [[Value]]: `exprValue`, [[Target]]: `empty`}.

13.15 The try Statement

Syntax

```
TryStatement[Yield, Await, Return] :
Finally[?Yield, ?Await, ?Return]
```

```
Catch[Yield, Await, Return] :
```

```
Finally[Yield, Await, Return] :
finally Block[?Yield, ?Await, ?Return]
```

```
CatchParameter[Yield, Await] :
  BindingIdentifier[?Yield, ?Await]
  BindingPattern[?Yield, ?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.

13.15.1 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`. 
13.15.2 Static Semantics: ContainsDuplicateLabels

With parameter labelSet.

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.

13.15.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter labelSet.

TryStatement : try Block Catch

  1. Let hasUndefinedLabels be ContainsUndefinedBreakTarget of Block with argument labelSet.
  2. If hasUndefinedLabels is true, return true.
  3. Return ContainsUndefinedBreakTarget of Catch with argument labelSet.

TryStatement : try Block Finally

  1. Let hasUndefinedLabels be ContainsUndefinedBreakTarget of Block with argument labelSet.
  2. If hasUndefinedLabels is true, return true.
3. Return ContainsUndefinedBreakTarget of Finally with argument labelSet.

TryStatement : try Block Catch Finally

1. Let hasUndefinedLabels be ContainsUndefinedBreakTarget of Block with argument labelSet.
2. If hasUndefinedLabels is true, return true.
3. Let hasUndefinedLabels be ContainsUndefinedBreakTarget of Catch with argument labelSet.
4. If hasUndefinedLabels is true, return true.
5. Return ContainsUndefinedBreakTarget of Finally with argument labelSet.

Catch : catch ( CatchParameter ) Block

1. Return ContainsUndefinedBreakTarget of Block with argument labelSet.

13.15.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters iterationSet and labelSet.

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 Contains Undefined Continue Target of \( Block \) with arguments \( \text{iterationSet} \) and « ».

### 13.15.5 Static Semantics: VarDeclaredNames

\( \text{TryStatement} : \text{try} \ Block \ Catch \)

1. Let \( names \) be VarDeclaredNames of \( Block \).
2. Append to \( names \) the elements of the VarDeclaredNames of \( Catch \).
3. Return \( names \).

\( \text{TryStatement} : \text{try} \ Block \ Finally \)

1. Let \( names \) be VarDeclaredNames of \( Block \).
2. Append to \( names \) the elements of the VarDeclaredNames of \( Finally \).
3. Return \( names \).

\( \text{TryStatement} : \text{try} \ Block \ Catch \ Finally \)

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 \( names \).

\( \text{Catch} : \text{catch} ( \ CatchParameter \ ) \ Block \)

1. Return the VarDeclaredNames of \( Block \).

### 13.15.6 Static Semantics: VarScopedDeclarations

\( \text{TryStatement} : \text{try} \ Block \ Catch \)

1. Let \( declarations \) be VarScopedDeclarations of \( Block \).
2. Append to \( declarations \) the elements of the VarScopedDeclarations of \( Catch \).
3. Return \( declarations \).

\( \text{TryStatement} : \text{try} \ Block \ Finally \)

1. Let \( declarations \) be VarScopedDeclarations of \( Block \).
2. Append to \( declarations \) the elements of the VarScopedDeclarations of \( Finally \).
3. Return \( declarations \).

\( \text{TryStatement} : \text{try} \ Block \ Catch \ Finally \)

1. Let \( declarations \) be VarScopedDeclarations of \( Block \).
2. Append to \( declarations \) the elements of the VarScopedDeclarations of \( Catch \).
3. Append to declarations the elements of the VarScopedDeclarations of Finally.
4. Return declarations.

Catch: catch (CatchParameter) Block

1. Return the VarScopedDeclarations of Block.

13.15.7 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. Let catchEnvRec be catchEnv's EnvironmentRecord.
4. For each element argName of the BoundNames of CatchParameter, do
   a. Perform ! catchEnvRec.CreateMutableBinding(argName, false).
5. Set the running execution context's LexicalEnvironment to catchEnv.
6. Let status be the result of performing BindingInitialization for CatchParameter passing thrownValue and catchEnv as arguments.
7. If status is an abrupt completion, then
   a. Set the running execution context's LexicalEnvironment to oldEnv.
   b. Return Completion(status).
8. Let B be the result of evaluating Block.
9. Set the running execution context's LexicalEnvironment to oldEnv.
10. Return Completion(B).

NOTE No matter how control leaves the Block the LexicalEnvironment is always restored to its former state.

13.15.8 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 parameter B.[[Value]].
3. Else, let C be B.
4. Return Completion(UpdateTimeEmpty(C, undefined)).

TryStatement : try Block Finally

1. Let B be the result of evaluating Block.
2. Let \( F \) be the result of evaluating \( \text{Finally} \).
3. If \( F.\[\text{Type}\] \) is \text{normal}, set \( F \) to \( B \).
4. Return \( \text{Completion} \left( \text{UpdateEmpty} \left( F, \text{undefined} \right) \right) \).

\text{TryStatement} : \text{try} \ Block \ \text{Catch} \ \text{Finally}

1. Let \( B \) be the result of evaluating \( \text{Block} \).
2. If \( B.\[\text{Type}\] \) is \text{throw}, let \( C \) be \( \text{CatchClauseEvaluation of} \ \text{Catch} \) with parameter \( B.\[\text{Value}\] \).
3. Else, let \( C \) be \( B \).
4. Let \( F \) be the result of evaluating \( \text{Finally} \).
5. If \( F.\[\text{Type}\] \) is \text{normal}, set \( F \) to \( C \).
6. Return \( \text{Completion} \left( \text{UpdateEmpty} \left( F, \text{undefined} \right) \right) \).

13.16 The debugger Statement

\text{Syntax}

\[ DebuggerStatement : \text{debugger} ; \]

13.16.1 Runtime Semantics: Evaluation

\text{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.

\( DebuggerStatement : \text{debugger} ; \)

1. If an implementation-defined debugging facility is available and enabled, then
   a. Perform an implementation-defined debugging action.
   b. Let \( result \) be an implementation-defined \( \text{Completion} \) value.
2. Else,
   a. Let \( result \) be \( \text{NormalCompletion} \left( \text{empty} \right) \).
3. Return \( result \).

14 ECMAScript Language: Functions and Classes

\text{NOTE} Various ECMAScript language elements cause the creation of ECMAScript function objects (9.2). Evaluation of such functions starts with the execution of their \([\text{[[Call]]}] \) internal
14.1 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] 
  }

UniqueFormalParameters[Yield, Await] :
  FormalParameters[Yield, Await]

FormalParameters[Yield, Await] :
  [empty]
  FunctionRestParameter[Yield, Await]
  FormalParameterList[Yield, Await]
  FormalParameterList[Yield, Await] ,
  FormalParameterList[Yield, Await] , FunctionRestParameter[Yield, Await]

FormalParameterList[Yield, Await] :
  FormalParameter[Yield, Await]
  FormalParameterList[Yield, Await] , FormalParameter[Yield, Await]

FunctionRestParameter[Yield, Await] :
  BindingRestElement[Yield, Await]

FormalParameter[Yield, Await] :
  BindingElement[Yield, Await]

FunctionBody[Yield, Await] :
  FunctionStatementList[Yield, Await]

FunctionStatementList[Yield, Await] :


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. A Directive Prologue may be an empty sequence.

A Use Strict Directive is an ExpressionStatement in a Directive Prologue whose StringLiteral is either the exact code unit sequences "use strict" or 'use strict'. A Use Strict Directive may not contain an EscapeSequence or LineContinuation.

A Directive Prologue may contain more than one Use Strict Directive. However, an implementation may issue a warning if this occurs.

NOTE

The 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.

14.1.2 Static Semantics: Early Errors

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
FunctionDeclaration : function ( FormalParameters ) { FunctionBody }
FunctionExpression : function BindingIdentifier ( FormalParameters ) { FunctionBody }

- If the source code matching this production is strict mode code, the Early Error rules for UniqueFormalParameters : FormalParameters are applied.
- If the source code matching this production is strict mode code, it is a Syntax Error if BindingIdentifier is the IdentifierName eval or the IdentifierName arguments.
- It is a Syntax Error if ContainsUseStrict 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 1 The LexicallyDeclaredNames of a FunctionBody does not include identifiers bound using var or function declarations.

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 2 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.

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.

14.1.3 Static Semantics: BoundNames

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }

1. Return the BoundNames of BindingIdentifier.

FunctionDeclaration : function ( FormalParameters ) { FunctionBody }

1. Return « "*default*" ».

NOTE "*default*" is used within this specification as a synthetic name for hoistable anonymous functions that are defined using export declarations.

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.

### 14.1.4 Static Semantics: Contains

With parameter symbol.

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
FunctionDeclaration : function ( FormalParameters ) { FunctionBody }
FunctionExpression : function BindingIdentifier ( FormalParameters ) { FunctionBody }

1. Return false.

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

### 14.1.5 Static Semantics: ContainsExpression

FormalParameters : [empty]

1. Return false.

FormalParameters : FormalParameterList , FunctionRestParameter

1. If ContainsExpression of FormalParameterList is true, return true.
2. Return ContainsExpression of FunctionRestParameter.

FormalParameterList : FormalParameterList , FormalParameter

1. If ContainsExpression of FormalParameterList is true, return true.
2. Return ContainsExpression of FormalParameter.

### 14.1.6 Static Semantics: ContainsUseStrict
FunctionBody : FunctionStatementList

1. If the Directive Prologue of FunctionStatementList contains a Use Strict Directive, return true; otherwise, return false.

14.1.7 Static Semantics: ExpectedArgumentCount

FormalParameters : [empty]

1. Return 0.

FormalParameters : FormalParameterList , FunctionRestParameter

1. Return ExpectedArgumentCount of FormalParameterList.

NOTE The ExpectedArgumentCount of a FormalParameterList is the number of FormalParameters to the left of either the rest parameter or the first FormalParameter with an Initializer. A FormalParameter without an initializer is allowed after the first parameter with an initializer but such parameters are considered to be optional with undefined as their default value.

FormalParameterList : FormalParameterList , FormalParameter

1. Let count be ExpectedArgumentCount of FormalParameterList.
2. If HasInitializer of FormalParameterList is true or HasInitializer of FormalParameter is true, return count.

14.1.8 Static Semantics: HasInitializer

FormalParameterList : FormalParameterList , FormalParameter

1. If HasInitializer of FormalParameterList is true, return true.
2. Return HasInitializer of FormalParameter.

14.1.9 Static Semantics: HasName

FunctionExpression : function ( FormalParameters ) { FunctionBody }

1. Return false.

FunctionExpression : function BindingIdentifier ( FormalParameters ) { FunctionBody }

1. Return true.
14.1.10 Static Semantics: IsAnonymousFunctionDefinition ( expr )

The abstract operation IsAnonymousFunctionDefinition determines if its argument is a function definition that does not bind a name. The argument expr is the result of parsing an AssignmentExpression orInitializer. The following steps are taken:

1. If IsFunctionDefinition of expr is false, return false.
2. Let hasName be the result of HasName of expr.
3. If hasName is true, return false.
4. Return true.

14.1.11 Static Semantics: IsConstantDeclaration

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }

FunctionDeclaration : function ( FormalParameters ) { FunctionBody }

1. Return false.

14.1.12 Static Semantics: IsFunctionDefinition

FunctionExpression : function BindingIdentifier ( FormalParameters ) { FunctionBody }

1. Return true.

14.1.13 Static Semantics: IsSimpleParameterList

FormalParameters : [empty]

1. Return true.

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.

14.1.14 Static Semantics: LexicallyDeclaredNames

FunctionStatementList : [empty]
1. Return a new empty `List`.

`FunctionStatementList : StatementList`

1. Return `TopLevelLexicallyDeclaredNames` of `StatementList`.

### 14.1.15 Static Semantics: LexicallyScopedDeclarations

`FunctionStatementList : [empty]`

1. Return a new empty `List`.

`FunctionStatementList : StatementList`

1. Return `TopLevelLexicallyScopedDeclarations` of `StatementList`.

### 14.1.16 Static Semantics: VarDeclaredNames

`FunctionStatementList : [empty]`

1. Return a new empty `List`.

`FunctionStatementList : StatementList`

1. Return `TopLevelVarDeclaredNames` of `StatementList`.

### 14.1.17 Static Semantics: VarScopedDeclarations

`FunctionStatementList : [empty]`

1. Return a new empty `List`.

`FunctionStatementList : StatementList`

1. Return `TopLevelVarScopedDeclarations` of `StatementList`.

### 14.1.18 Runtime Semantics: EvaluateBody

With parameters `functionObject` and `List argumentsList`.

`FunctionBody : FunctionStatementList`

1. Perform ? `FunctionDeclarationInstantiation(functionObject, argumentsList)`.  
2. Return the result of evaluating `FunctionStatementList`. 
14.1.19 Runtime Semantics: IteratorBindingInitialization

With parameters `iteratorRecord` and `environment`.

NOTE 1 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.

**FormalParameters** : [empty]

1. Return `NormalCompletion`(`empty`).

**FormalParameters** : `FormalParameterList` , `FunctionRestParameter`

1. Let `restIndex` be the result of performing `IteratorBindingInitialization` for `FormalParameterList` using `iteratorRecord` and `environment` as the arguments.
2. `ReturnIfAbrupt`(`restIndex`).
3. Return the result of performing `IteratorBindingInitialization` for `FunctionRestParameter` using `iteratorRecord` and `environment` as the arguments.

**FormalParameterList** : `FormalParameterList` , `FormalParameter`

1. Let `status` be the result of performing `IteratorBindingInitialization` for `FormalParameterList` using `iteratorRecord` and `environment` as the arguments.
2. `ReturnIfAbrupt`(`status`).
3. Return the result of performing `IteratorBindingInitialization` for `FormalParameter` using `iteratorRecord` and `environment` as the arguments.

**FormalParameter** : `BindingElement`

1. If `ContainsExpression` of `BindingElement` is `false`, return the result of performing `IteratorBindingInitialization` for `BindingElement` using `iteratorRecord` and `environment` as the arguments.
2. Let `currentContext` be the running execution context.
3. Let `originalEnv` be the VariableEnvironment of `currentContext`.
4. Assert: The VariableEnvironment and LexicalEnvironment of `currentContext` are the same.
5. Assert: `environment` and `originalEnv` are the same.
6. Let `paramVarEnv` be `NewDeclarativeEnvironment`(`originalEnv`).
7. Set the VariableEnvironment of `currentContext` to `paramVarEnv`.
8. Set the LexicalEnvironment of `currentContext` to `paramVarEnv`.
9. Let `result` be the result of performing `IteratorBindingInitialization` for `BindingElement` using `iteratorRecord` and `environment` as the arguments.
10. Set the VariableEnvironment of `currentContext` to `originalEnv`. 
11. Set the LexicalEnvironment of `currentContext` to `originalEnv`.
12. Return `result`.

**NOTE 2** The new Environment Record created in step 6 is only used if the BindingElement contains a direct eval.

**FunctionRestParameter : BindingRestElement**

1. If ContainsExpression of BindingRestElement is `false`, return the result of performing IteratorBindingInitialization for BindingRestElement using `iteratorRecord` and `environment` as the arguments.
2. Let `currentContext` be the running execution context.
3. Let `originalEnv` be the VariableEnvironment of `currentContext`.
4. Assert: The VariableEnvironment and LexicalEnvironment of `currentContext` are the same.
5. Assert: `environment` and `originalEnv` are the same.
6. Let `paramVarEnv` be NewDeclarativeEnvironment(`originalEnv`).
7. Set the VariableEnvironment of `currentContext` to `paramVarEnv`.
8. Set the LexicalEnvironment of `currentContext` to `paramVarEnv`.
9. Let `result` be the result of performing IteratorBindingInitialization for BindingRestElement using `iteratorRecord` and `environment` as the arguments.
10. Set the VariableEnvironment of `currentContext` to `originalEnv`.
11. Set the LexicalEnvironment of `currentContext` to `originalEnv`.
12. Return `result`.

**NOTE 3** The new Environment Record created in step 6 is only used if the BindingRestElement contains a direct eval.

**14.1.20 Runtime Semantics: InstantiateFunctionObject**

With parameter `scope`.

**FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }**

1. If the function code for FunctionDeclaration is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `name` be StringValue of BindingIdentifier.
4. Perform MakeConstructor(`F`).
5. Perform SetFunctionName(`F`, `name`).

**FunctionDeclaration : function ( FormalParameters ) { FunctionBody }**
1. Let $F$ be `FunctionCreate(Normal, FormalParameters, FunctionBody, scope, true).
2. Perform `MakeConstructor(F).
3. Perform `SetFunctionName(F, "default").
4. 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.

### 14.1.21 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 ( FormalParameters ) { FunctionBody }`

1. If the function code for `FunctionExpression` is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `scope` be the LexicalEnvironment of the running execution context.
3. Let `closure` be `FunctionCreate(Normal, FormalParameters, FunctionBody, scope, strict)`.
4. Perform `MakeConstructor(closure)`.
5. Return `closure`.

**FunctionExpression** : `function BindingIdentifier ( FormalParameters ) { FunctionBody }`

1. If the function code for `FunctionExpression` is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `scope` be the running execution context's LexicalEnvironment.
3. Let `funcEnv` be `NewDeclarativeEnvironment(scope)`.
4. Let `envRec` be `funcEnv`'s EnvironmentRecord.
5. Let `name` be StringValue of `BindingIdentifier`.
6. Perform `envRec.CreateImmutableBinding(name, false)`.
7. Let `closure` be `FunctionCreate(Normal, FormalParameters, FunctionBody, funcEnv, strict)`.
8. Perform `MakeConstructor(closure)`.
9. Perform `SetFunctionName(closure, name)`.
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`.

A `prototype` property is automatically created for every function defined using a `FunctionDeclaration` or `FunctionExpression`, to allow for the possibility that the function will be used as a constructor.

FunctionStatementList : [empty]

1. Return `NormalCompletion(undefined)`.

### 14.2 Arrow Function Definitions

#### Syntax

```
ArrowFunction[In, Yield, Await] :
  ArrowParameters[?Yield, ?Await] [no LineTerminator here] => ConciseBody[?In]
```

```
ArrowParameters[Yield, Await] :
  BindingIdentifier[?Yield, ?Await]
  CoverParenthesizedExpressionAndArrowParameterList[?Yield, ?Await]
```

```
ConciseBody[In] :
  [lookahead ≠ {} AssignmentExpression[?In, ~Yield, ~Await]
  { FunctionBody[~Yield, ~Await] }
```

#### Supplemental Syntax

When the production

```
ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList
```

is recognized the following grammar is used to refine the interpretation of `CoverParenthesizedExpressionAndArrowParameterList`:

```
ArrowFormalParameters[Yield, Await] :
  ( UniqueFormalParameters[?Yield, ?Await] )
```

### 14.2.1 Static Semantics: Early Errors

```
ArrowFunction : ArrowParameters => ConciseBody
```
It is a Syntax Error if `ArrowParameters` Contains `YieldExpression` is `true`.

It is a Syntax Error if `ArrowParameters` Contains `AwaitExpression` is `true`.

It is a Syntax Error if `ContainsUseStrict of ConciseBody` is `true` and `IsSimpleParameterList of ArrowParameters` is `false`.

It is a Syntax Error if any element of the `BoundNames` of `ArrowParameters` also occurs in the `LexicallyDeclaredNames` of `ConciseBody`.

ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList

- It is a Syntax Error if the lexical token sequence matched by `CoverParenthesizedExpressionAndArrowParameterList` cannot be parsed with no tokens left over using `ArrowFormalParameters` as the goal symbol with its `Yield` and `Await` parameters set to the values used when parsing this `CoverParenthesizedExpressionAndArrowParameterList`.
- All early error rules for `ArrowFormalParameters` and its derived productions also apply to `CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList`.

### 14.2.2 Static Semantics: BoundNames

**ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList**

1. Let `formals` be `CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList`.
2. Return the `BoundNames` of `formals`.

### 14.2.3 Static Semantics: Contains

With parameter `symbol`.

**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`.

**NOTE** Normally, Contains does not look inside most function forms. However, Contains is used to detect `new.target`, `this`, and `super` usage within an ArrowFunction.

**ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList**

1. Let `formals` be `CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList`.
2. Return `formals Contains symbol`.

### 14.2.4 Static Semantics: ContainsExpression

**ArrowParameters : BindingIdentifier**
1. Return `false`.

### 14.2.5 Static Semantics: ContainsUseStrict

*ConciseBody : AssignmentExpression*

1. Return `false`.

### 14.2.6 Static Semantics: ExpectedArgumentCount

*ArrowParameters : BindingIdentifier*

1. Return 1.

### 14.2.7 Static Semantics: HasName

*ArrowFunction : ArrowParameters => ConciseBody*

1. Return `false`.

### 14.2.8 Static Semantics: IsSimpleParameterList

*ArrowParameters : BindingIdentifier*

1. Return `true`.

*ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList*

1. Let `formals` be CoveredFormalsList of `CoverParenthesizedExpressionAndArrowParameterList`.
2. Return IsSimpleParameterList of `formals`.

### 14.2.9 Static Semantics: CoveredFormalsList

*ArrowParameters : BindingIdentifier*

1. Return this `ArrowParameters`.

*CoverParenthesizedExpressionAndArrowParameterList[Yield, Await] :

- ( Expression )
- ( )
- ( ... BindingIdentifier )
- ( ... BindingPattern )
- ( Expression , ... BindingIdentifier )
- ( Expression , ... BindingPattern )
1. Return the result of parsing the lexical token stream matched by
   CoverParenthesizedExpressionAndArrowParameterList using ArrowFormalParameters as the goal
   symbol with its [Yield] and [Await] parameters set to the values used when parsing this
   CoverParenthesizedExpressionAndArrowParameterList.

14.2.10 Static Semantics: LexicallyDeclaredNames
ConciseBody : AssignmentExpression

1. Return a new empty List.

14.2.11 Static Semantics: LexicallyScopedDeclarations
ConciseBody : AssignmentExpression

1. Return a new empty List.

14.2.12 Static Semantics: VarDeclaredNames
ConciseBody : AssignmentExpression

1. Return a new empty List.

14.2.13 Static Semantics: VarScopedDeclarations
ConciseBody : AssignmentExpression

1. Return a new empty List.

14.2.14 Runtime Semantics: IteratorBindingInitialization

With parameters iteratorRecord and environment.

NOTE When undefined is passed for environment it indicates that a PutValue operation should
be used to assign the initialization value. This is the case for formal parameter lists of non-
strict functions. In that case the formal parameter bindings are preinitialized in order to
deal with the possibility of multiple parameters with the same name.

ArrowParameters : BindingIdentifier

1. Assert: iteratorRecord.[[Done]] is false.
2. Let next be IteratorStep(iteratorRecord.[[Iterator]]).
3. If next is an abrupt completion, set iteratorRecord.[[Done]] to true.
4. ReturnIfAbrupt(next).
5. If \( \text{next} \) is \text{false}, set \( \text{iteratorRecord.}[[\text{Done}]] \) to \text{true}.

6. Else,
   a. Let \( \nu \) be \( \text{IteratorValue(} \text{next} \text{)} \).
   b. If \( \nu \) is an abrupt completion, set \( \text{iteratorRecord.}[[\text{Done}]] \) to \text{true}.
   c. \text{ReturnIfAbrupt(} \nu \text{)}.

7. If \( \text{iteratorRecord.}[[\text{Done}]] \) is \text{true}, let \( \nu \) be \text{undefined}.

8. Return the result of performing BindingInitialization for \( \text{BindingIdentifier} \) using \( \nu \) and \( \text{environment} \) as the arguments.

### 14.2.15 Runtime Semantics: EvaluateBody

With parameters \( \text{functionObject} \) and \( \text{List argumentsList} \).

\[
\text{ConciseBody : AssignmentExpression}
\]

1. Perform ? \( \text{FunctionDeclarationInstantiation(functionObject, argumentsList)} \).
2. Let \( \text{exprRef} \) be the result of evaluating \( \text{AssignmentExpression} \).
3. Let \( \text{exprValue} \) be ? \( \text{GetValue(exprRef)} \).
4. Return \( \text{Completion}([[\text{Type}]]: \text{return}, [[\text{Value}]]: \text{exprValue}, [[\text{Target}]]: \text{empty}] \).

### 14.2.16 Runtime Semantics: Evaluation

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

1. If the function code for this \( \text{ArrowFunction} \) is \text{strict mode code}, let \( \text{strict} \) be \text{true}. Otherwise let \( \text{strict} \) be \text{false}.
2. Let \( \text{scope} \) be the LexicalEnvironment of the running execution context.
3. Let \( \text{parameters} \) be \( \text{CoveredFormalsList} \) of \( \text{ArrowParameters} \).
4. Let \( \text{closure} \) be \( \text{FunctionCreate(Arrow, parameters, ConciseBody, scope, strict)} \).
5. Return \( \text{closure} \).

**NOTE**

An \( \text{ArrowFunction} \) does not define local bindings for \( \text{arguments, super, this, or new.target} \). Any reference to \( \text{arguments, super, this, or new.target} \) within an \( \text{ArrowFunction} \) 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 \( \text{ArrowFunction} \) may contain references to \( \text{super} \), the function object created in step 4 is not made into a method by performing \( \text{MakeMethod} \). An \( \text{ArrowFunction} \) that references \( \text{super} \) is always contained within a non-\( \text{ArrowFunction} \) and the necessary state to implement \( \text{super} \) is accessible via the \( \text{scope} \) that is captured by the function object of the \( \text{ArrowFunction} \).
14.3 Method Definitions

Syntax

\[
\text{MethodDefinition}[\text{Yield, Await}] : \\
\text{PropertyName}[\text{?Yield, ?Await}] (\text{UniqueFormalParameters}[\text{~Yield, ~Await}]) \{ \\
\text{FunctionBody}[\text{~Yield, ~Await}] \}
\]

\[
\text{GeneratorMethod}[\text{?Yield, ?Await}] \\
\text{AsyncMethod}[\text{?Yield, ?Await}] \\
\text{get} \ \text{PropertyName}[\text{?Yield, ?Await}] ( ) \{ \text{FunctionBody}[\text{~Yield, ~Await}] \} \\
\text{set} \ \text{PropertyName}[\text{?Yield, ?Await}] (\text{PropertySetParameterList}) \{ \\
\text{FunctionBody}[\text{~Yield, ~Await}] \}
\]

\[
\text{PropertySetParameterList} : \\
\text{FormalParameter}[\text{~Yield, ~Await}]
\]

14.3.1 Static Semantics: Early Errors

\[
\text{MethodDefinition} : \text{PropertyName} (\text{UniqueFormalParameters}) \{ \text{FunctionBody} \}
\]

- It is a Syntax Error if \text{ContainsUseStrict} of \text{FunctionBody} is \text{true} and \text{IsSimpleParameterList} of \text{UniqueFormalParameters} is \text{false}.
- It is a Syntax Error if any element of the \text{BoundNames} of \text{UniqueFormalParameters} also occurs in the \text{LexicallyDeclaredNames} of \text{FunctionBody}.

\[
\text{MethodDefinition} : \text{set} \ \text{PropertyName} (\text{PropertySetParameterList}) \{ \text{FunctionBody} \}
\]

- It is a Syntax Error if \text{BoundNames} of \text{PropertySetParameterList} contains any duplicate elements.
- It is a Syntax Error if \text{ContainsUseStrict} of \text{FunctionBody} is \text{true} and \text{IsSimpleParameterList} of \text{PropertySetParameterList} is \text{false}.
- It is a Syntax Error if any element of the \text{BoundNames} of \text{PropertySetParameterList} also occurs in the \text{LexicallyDeclaredNames} of \text{FunctionBody}.

14.3.2 Static Semantics: ComputedPropertyContains

With parameter \text{symbol}.

\[
\text{MethodDefinition} : \\
\text{PropertyName} (\text{UniqueFormalParameters}) \{ \text{FunctionBody} \} \\
\text{get} \ \text{PropertyName} ( ) \{ \text{FunctionBody} \} \\
\text{set} \ \text{PropertyName} (\text{PropertySetParameterList}) \{ \text{FunctionBody} \}
\]
1. Return the result of ComputedPropertyContains for PropertyName with argument symbol.

### 14.3.3 Static Semantics: ExpectedArgumentCount

PropertySetParameterList : FormalParameter

1. If HasInitializer of FormalParameter is true, return 0.
2. Return 1.

### 14.3.4 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.

### 14.3.5 Static Semantics: PropName

MethodDefinition :

PropertyName ( UniqueFormalParameters ) { FunctionBody }
get PropertyName ( ) { FunctionBody }
set PropertyName ( PropertySetParameterList ) { FunctionBody }

1. Return PropName of PropertyName.

### 14.3.6 Static Semantics: SpecialMethod

MethodDefinition : PropertyName ( UniqueFormalParameters ) { FunctionBody }

1. Return false.
1. Return `true`.

### 14.3.7 Runtime Semantics: DefineMethod

With parameters `object` and optional parameter `functionPrototype`.

\[
\text{MethodDefinition} : \text{PropertyName} ( \text{UniqueFormalParameters} ) \{ \text{FunctionBody} \}
\]

1. Let `propKey` be the result of evaluating `PropertyName`.
2. `ReturnIfAbrupt(propKey)`.
3. If the function code for this `MethodDefinition` is `strict mode` code, let `strict` be `true`. Otherwise let `strict` be `false`.
4. Let `scope` be the running execution context's LexicalEnvironment.
5. If `functionPrototype` was passed as a parameter, then
   a. Let `kind` be `Normal`.
   b. Let `prototype` be `functionPrototype`.
6. Else,
   a. Let `kind` be `Method`.
   b. Let `prototype` be the intrinsic object `%FunctionPrototype%`.
7. Let `closure` be `FunctionCreate(kind, UniqueFormalParameters, FunctionBody, scope, strict, prototype)`.
8. Perform `MakeMethod(closure, object)`.
9. Return the `Record` `[[Key]]: propKey, [[Closure]]: closure`.

### 14.3.8 Runtime Semantics: PropertyDefinitionEvaluation

With parameters `object` and `enumerable`.

\[
\text{MethodDefinition} : \text{PropertyName} ( \text{UniqueFormalParameters} ) \{ \text{FunctionBody} \}
\]

1. Let `methodDef` be `DefineMethod` of `MethodDefinition` with argument `object`.
2. `ReturnIfAbrupt(methodDef)`.
3. Perform `SetFunctionName(methodDef.[[Closure]], methodDef.[[Key]])`.
4. Let `desc` be the PropertyDescriptor `[[Value]]: methodDef.[[Closure]], [[Writable]]: true, [[Enumerable]]: enumerable, [[Configurable]]: true`.
5. Return `DefinePropertyOrThrow(object, methodDef.[[Key]], desc)`.

\[
\text{MethodDefinition} : \text{get} \ \text{PropertyName} ( \ ) \{ \text{FunctionBody} \}
\]

1. Let `propKey` be the result of evaluating `PropertyName`.
2. `ReturnIfAbrupt(propKey)`.
3. If the function code for this `MethodDefinition` is `strict mode` code, let `strict` be `true`. Otherwise let `strict` be `false`.
4. Let `scope` be the running execution context's LexicalEnvironment.
5. Let `formalParameterList` be an instance of the production `FormalParameters : [empty]`.
7. Perform `MakeMethod(closure, object)`.
8. Perform `SetFunctionName(closure, propKey, "get")`.
9. Let `desc` be the PropertyDescriptor([[Get]]: closure, [[Enumerable]]: enumerable, [[Configurable]]: true).

```
MethodDefinition : set PropertyName ( PropertySetParameterList ) { FunctionBody }
```

1. Let `propKey` be the result of evaluating `PropertyName`.
2. ReturnIfAbrupt(`propKey`).
3. If the function code for this `MethodDefinition` is strict mode code, let `strict` be true. Otherwise let `strict` be false.
4. Let `scope` be the running execution context's LexicalEnvironment.
5. Let `closure` be `FunctionCreate(Method, PropertySetParameterList, FunctionBody, scope, strict)`.
6. Perform `MakeMethod(closure, object)`.
7. Perform `SetFunctionName(closure, propKey, "set")`.
8. Let `desc` be the PropertyDescriptor([[Set]]: closure, [[Enumerable]]: enumerable, [[Configurable]]: true).

### 14.4 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 : 
```
Function Body [+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 25.3.3.

14.4.1 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 ContainsUseStrict 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 }
GeneratorDeclaration : function * ( FormalParameters ) { GeneratorBody }
GeneratorExpression : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

- If the source code matching this production is strict mode code, the Early Error rules for UniqueFormalParameters : FormalParameters are applied.
- If the source code matching this production is strict mode code, it is a Syntax Error if BindingIdentifier is the IdentifierName eval or the IdentifierName arguments.
- It is a Syntax Error if ContainsUseStrict of GeneratorBody is true and IsSimpleParameterList of FormalParameters is false.
- It is a Syntax Error if any element of the BoundNames of FormalParameters also occurs in the LexicallyDeclaredNames of GeneratorBody.
- It is a Syntax Error if FormalParameters Contains YieldExpression is true.
- It is a Syntax Error if FormalParameters Contains SuperProperty is true.
- It is a Syntax Error if GeneratorBody Contains SuperProperty is true.
- It is a Syntax Error if FormalParameters Contains SuperCall is true.
It is a Syntax Error if `GeneratorBody Contains SuperCall is true`.

### 14.4.2 Static Semantics: BoundNames

**GeneratorDeclaration**: `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. Return the BoundNames of `BindingIdentifier`.

**GeneratorDeclaration**: `function * ( FormalParameters ) { GeneratorBody }`

1. Return «"*default*"».

**NOTE** "*default*" is used within this specification as a synthetic name for hoistable anonymous functions that are defined using export declarations.

### 14.4.3 Static Semantics: ComputedPropertyContains

With parameter `symbol`.

**GeneratorMethod**: `* PropertyName ( UniqueFormalParameters ) { GeneratorBody }`

1. Return the result of ComputedPropertyContains for `PropertyName with argument symbol`.

### 14.4.4 Static Semantics: Contains

With parameter `symbol`.

**GeneratorDeclaration**: `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

**GeneratorDeclaration**: `function * ( FormalParameters ) { GeneratorBody }`

**GeneratorExpression**: `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. Return `false`.

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

### 14.4.5 Static Semantics: HasDirectSuper

**GeneratorMethod**: `* PropertyName ( UniqueFormalParameters ) { GeneratorBody }`

1. If `UniqueFormalParameters Contains SuperCall is true`, return `true`.
2. Return `GeneratorBody Contains SuperCall`.

### 14.4.6 Static Semantics: HasName
GeneratorExpression : `function * ( FormalParameters ) { GeneratorBody }`

1. Return `false`.

GeneratorExpression : `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. Return `true`.

14.4.7 Static Semantics: IsConstantDeclaration

GeneratorDeclaration : `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`
GeneratorDeclaration : `function * ( FormalParameters ) { GeneratorBody }`

1. Return `false`.

14.4.8 Static Semantics: IsFunctionDefinition

GeneratorExpression : `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. Return `true`.

14.4.9 Static Semantics: PropName

GeneratorMethod : * PropertyName ( UniqueFormalParameters ) { GeneratorBody }

1. Return PropName of `PropertyName`.

14.4.10 Runtime Semantics: EvaluateBody

With parameters `functionObject` and `List argumentsList`.

`GeneratorBody : FunctionBody`

1. Perform `FunctionDeclarationInstantiation(functionObject, argumentsList)`.
2. Let `G` be `OrdinaryCreateFromConstructor(functionObject, "%GeneratorPrototype%", « [[GeneratorState]], [[GeneratorContext]] »)`.
3. Perform `GeneratorStart(G, FunctionBody)`.
4. Return `Completion{[[Type]]: return, [[Value]]: G, [[Target]]: empty}`.

14.4.11 Runtime Semantics: InstantiateFunctionObject

With parameter `scope`.

GeneratorDeclaration : `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`
1. If the function code for \textit{GeneratorDeclaration} is \textit{strict mode code}, let \textit{strict} be \textit{true}. Otherwise let \textit{strict} be \textit{false}.

2. Let \textit{name} be StringValue of \textit{BindingIdentifier}.

3. Let \textit{F} be \textit{GeneratorFunctionCreate}(Normal, FormalParameters, GeneratorBody, scope, \textit{strict}).

4. Let \textit{prototype} be \textit{ObjectCreate}(\%GeneratorPrototype\%).

5. Perform \textit{DefinePropertyOrThrow}(\textit{F}, "\textit{prototype}"), PropertyDescriptor([[Value]: \textit{prototype},
   [[Writable]]: \textit{true}, [[Enumerable]]: \textit{false}, [[Configurable]]: \textit{false}]).

6. Perform \textit{SetFunctionName}(\textit{F}, \textit{name}).

7. Return \textit{F}.

\textit{GeneratorDeclaration} : \textbf{function} * ( FormaliParameters ) \{ GeneratorBody \}

1. Let \textit{F} be \textit{GeneratorFunctionCreate}(Normal, FormalParameters, GeneratorBody, scope, \textit{true}).

2. Let \textit{prototype} be \textit{ObjectCreate}(\%GeneratorPrototype\%).

3. Perform \textit{DefinePropertyOrThrow}(\textit{F}, "\textit{prototype}"), PropertyDescriptor([[Value]: \textit{prototype},
   [[Writable]]: \textit{true}, [[Enumerable]]: \textit{false}, [[Configurable]]: \textit{false}]).

4. Perform \textit{SetFunctionName}(\textit{F}, "\textit{default}").

5. Return \textit{F}.

\textbf{NOTE} An anonymous \textit{GeneratorDeclaration} can only occur as part of an \texttt{export default} declaration, and its function code is therefore always \textit{strict mode code}.

\subsection*{14.4.12 Runtime Semantics: PropertyDefinitionEvaluation}

With parameters \textit{object} and \textit{enumerable}.

\textit{GeneratorMethod} : * \ PropertyName ( UniqueFormaliParameters ) \{ GeneratorBody \}

1. Let \textit{propKey} be the result of evaluating \textit{PropertyName}.

2. ReturnIfAbrupt(\textit{propKey}).

3. If the function code for this \textit{GeneratorMethod} is \textit{strict mode code}, let \textit{strict} be \textit{true}. Otherwise let \textit{strict} be \textit{false}.

4. Let \textit{scope} be the running execution context's LexicalEnvironment.

5. Let \textit{closure} be \textit{GeneratorFunctionCreate}(Method, UniqueFormaliParameters, GeneratorBody, scope, \textit{strict}).

6. Perform \textit{MakeMethod}(\textit{closure}, \textit{object}).

7. Let \textit{prototype} be \textit{ObjectCreate}(\%GeneratorPrototype\%).

8. Perform \textit{DefinePropertyOrThrow}(\textit{closure}, "\textit{prototype}"), PropertyDescriptor([[Value]: \textit{prototype},
   [[Writable]]: \textit{true}, [[Enumerable]]: \textit{false}, [[Configurable]]: \textit{false}]).

9. Perform \textit{SetFunctionName}(\textit{closure}, \textit{propKey}).

10. Let \textit{desc} be the PropertyDescriptor([[Value]]: \textit{closure}, [[Writable]]: \textit{true}, [[Enumerable]]: \textit{enumerable},
    [[Configurable]]: \textit{true}).
11. Return \( \texttt{DefinePropertyOrThrow} \left( \text{object, propKey, desc} \right) \).

### 14.4.13 Runtime Semantics: Evaluation

#### GeneratorExpression : function * ( FormalParameters ) { GeneratorBody }

1. If the function code for this `GeneratorExpression` is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `scope` be the LexicalEnvironment of the running execution context.
3. Let `closure` be `GeneratorFunctionCreate`\((\text{Normal, FormalParameters, GeneratorBody, scope, strict})\).
4. Let `prototype` be `ObjectCreate`\(%\text{GeneratorPrototype}\%\).
5. Perform `DefinePropertyOrThrow`\((\text{closure, "prototype"}, \text{PropertyDescriptor}[[\text{Value}]: \text{prototype}, [[\text{Writable}]: \text{true}, [[\text{Enumerable}]: \text{false}, [[\text{Configurable}]: \text{false}}])\).

#### GeneratorExpression : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

1. If the function code for this `GeneratorExpression` is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `scope` be the running execution context's LexicalEnvironment.
3. Let `funcEnv` be `NewDeclarativeEnvironment`\((\text{scope})\).
4. Let `envRec` be `funcEnv`'s `EnvironmentRecord`.
5. Let `name` be `StringValue` of `BindingIdentifier`.
6. Perform `envRec`.CreateImmutableBinding\((\text{name, false})\).
7. Let `closure` be `GeneratorFunctionCreate`\((\text{Normal, FormalParameters, GeneratorBody, funcEnv, strict})\).
8. Let `prototype` be `ObjectCreate`\(%\text{GeneratorPrototype}\%\).
9. Perform `DefinePropertyOrThrow`\((\text{closure, "prototype"}, \text{PropertyDescriptor}[[\text{Value}]: \text{prototype}, [[\text{Writable}]: \text{true}, [[\text{Enumerable}]: \text{false}, [[\text{Configurable}]: \text{false}}])\).
10. Perform `SetFunctionName`\((\text{closure, name})\).
11. Perform \(\text{envRec}.\text{InitializeBinding} \left( \text{name, closure} \right)\).
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`.

### YieldExpression : yield

1. Return \( \texttt{GeneratorYield} \left( \text{CreateIterResultObject} \left( \text{undefined, false} \right) \right) \).

### YieldExpression : yield AssignmentExpression
1. Let `exprRef` be the result of evaluating `AssignmentExpression`.
2. Let `value` be `GetValue(exprRef)`.
3. Return `Generator Yield(CreateIteratorResultObject(value, false))`.

**YieldExpression : yield * AssignmentExpression**

1. Let `exprRef` be the result of evaluating `AssignmentExpression`.
2. Let `value` be `GetValue(exprRef)`.
3. Let `iterator` be `GetIterator(value)`.
4. Let `received` be `NormalCompletion(undefined)`.
5. Repeat,
   a. If `received. [[Type]]` is normal, then
      i. Let `innerResult` be `IteratorNext(iterator, received. [[Value]])`.
      ii. Let `done` be `IteratorComplete(innerResult)`.
      iii. If `done` is `true`, then
          1. Return `IteratorValue(innerResult)`.
          iv. Set `received` to `Generator Yield(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. NOTE: Exceptions from the inner iterator `throw` method are propagated. Normal completions from an inner `throw` method are processed similarly to an inner `next`.
          3. If `Type(innerResult)` is `Object`, throw a `TypeError` exception.
          4. Let `done` be `IteratorComplete(innerResult)`.
          5. If `done` is `true`, then
              a. Return `IteratorValue(innerResult)`.
          6. Set `received` to `Generator Yield(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. Perform `Iterator Close(iterator, Completion([[Type]]: normal, [[Value]]: empty, [[Target]]: empty))`.
          3. NOTE: The next step throws a `TypeError` to indicate that there was a `yield*` protocol violation: `iterator` does not have a `throw` method.
          4. Throw a `TypeError` exception.
   c. Else,
      i. Assert: `received. [[Type]]` is `return`.
      ii. Let `return` be `GetMethod(iterator, "return")`.
      iii. If `return` is `undefined`, return `Completion(received)`.
      iv. Let `innerReturnResult` be `Call(return, iterator, « received. [[Value]] »)`.
v. If `Type(innerReturnResult)` is not `Object`, throw a `TypeError` exception.

vi. Let `done` be `? IteratorComplete(innerReturnResult)`.

vii. If `done` is `true`, then

1. Let `value` be `? IteratorValue(innerReturnResult)`.

viii. Set `received` to `GeneratorYield(innerReturnResult)`.

14.5 Class Definitions

Syntax

```
ClassDeclaration[Yield, Await, Default] :
  class BindingIdentifier[?Yield, ?Await] ClassTail[?Yield, ?Await] [+Default] class ClassTail[?Yield, ?Await]

ClassExpression[Yield, Await] :
  class BindingIdentifier[?Yield, ?Await] opt ClassTail[?Yield, ?Await]

ClassTail[Yield, Await] :
  ClassHeritage[?Yield, ?Await] opt { ClassBody[?Yield, ?Await] opt }

ClassHeritage[Yield, Await] :
  extends LeftHandSideExpression[?Yield, ?Await]

ClassBody[Yield, Await] :
  ClassElementList[?Yield, ?Await]

ClassElementList[Yield, Await] :
  ClassElement[?Yield, ?Await]

ClassElement[Yield, Await] :
  MethodDefinition[?Yield, ?Await]
  static MethodDefinition[?Yield, ?Await]

```

| NOTE | A class definition is always strict mode code. |

14.5.1 Static Semantics: Early Errors

```
ClassTail : ClassHeritage { ClassBody }
```
• It is a Syntax Error if `ClassHeritage` is not present and the following algorithm evaluates to `true`:

  1. Let `constructor` be `ConstructorMethod` of `ClassBody`.
  2. If `constructor` is `empty`, return `false`.
  3. Return HasDirectSuper of `constructor`.

`ClassBody : ClassElementList`

• It is a Syntax Error if PrototypePropertyNameList of `ClassElementList` contains more than one occurrence of "`constructor`".

`ClassElement : MethodDefinition`

• It is a Syntax Error if PropName of `MethodDefinition` is not "`constructor`" and HasDirectSuper of `MethodDefinition` is `true`.
• It is a Syntax Error if PropName of `MethodDefinition` is "`constructor`" and SpecialMethod of `MethodDefinition` is `true`.

`ClassElement : static MethodDefinition`

• It is a Syntax Error if HasDirectSuper of `MethodDefinition` is `true`.
• It is a Syntax Error if PropName of `MethodDefinition` is "`prototype`".

### 14.5.2 Static Semantics: BoundNames

`ClassDeclaration : class BindingIdentifier ClassTail`

  1. Return the BoundNames of `BindingIdentifier`.

`ClassDeclaration : class ClassTail`

  1. Return "`*default*`".

### 14.5.3 Static Semantics: ConstructorMethod

`ClassElementList : ClassElement`

  1. If `ClassElement` is `ClassElement : ;`, return `empty`.
  2. If IsStatic of `ClassElement` is `true`, return `empty`.
  3. If PropName of `ClassElement` is not "`constructor`", return `empty`.
  4. Return `ClassElement`.

`ClassElementList : ClassElementList ClassElement`

  1. Let `head` be `ConstructorMethod` of `ClassElementList`.
  2. If `head` is not `empty`, return `head`. 
3. If `ClassElement` is `ClassElement : ;`, return `empty`.
4. If `IsStatic` of `ClassElement` is `true`, return `empty`.
5. If `PropName` of `ClassElement` is not "constructor", return `empty`.
6. Return `ClassElement`.

**NOTE**  
Early Error rules ensure that there is only one method definition named "constructor" and that it is not an accessor property or generator definition.

### 14.5.4 Static Semantics: `Contains`

**With parameter `symbol`.

```
ClassTail : ClassHeritage { ClassBody }
```

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

**NOTE**  
Static semantic rules that depend upon substructure generally do not look into class bodies except for `PropertyNames`.

### 14.5.5 Static Semantics: `ComputedPropertyContains`

**With parameter `symbol`.

```
ClassElementList : ClassElementList ClassElement
```

1. Let `inList` be the result of ComputedPropertyContains for `ClassElementList` with argument `symbol`.
2. If `inList` is `true`, return `true`.
3. Return the result of ComputedPropertyContains for `ClassElement` with argument `symbol`.

```
ClassElement : MethodDefinition
```

1. Return the result of ComputedPropertyContains for `MethodDefinition` with argument `symbol`.

```
ClassElement : static MethodDefinition
```

1. Return the result of ComputedPropertyContains for `MethodDefinition` with argument `symbol`.

```
ClassElement : ;
```
1. Return `false`.

### 14.5.6 Static Semantics: HasName
*ClassExpression : class ClassTail*

1. Return `false`.

*ClassExpression : class BindingIdentifier ClassTail*

1. Return `true`.

### 14.5.7 Static Semantics: IsConstantDeclaration
*ClassDeclaration : class BindingIdentifier ClassTail*

*ClassDeclaration : class ClassTail*

1. Return `false`.

### 14.5.8 Static Semantics: IsFunctionDefinition
*ClassExpression : class BindingIdentifier ClassTail*

1. Return `true`.

### 14.5.9 Static Semantics: IsStatic
*ClassElement : MethodDefinition*

1. Return `false`.

*ClassElement : static MethodDefinition*

1. Return `true`.

*ClassElement : ;*

1. Return `false`.

### 14.5.10 Static Semantics: NonConstructorMethodDefinitions
*ClassElementList : ClassElement*

1. If `ClassElement` is `ClassElement : ;`, return a new empty `List`.
2. If `IsStatic` of `ClassElement` is `false` and `PropName` of `ClassElement` is "constructor", return a new empty `List`. 
3. Return a List containing ClassElement.

ClassElementList : ClassElementList ClassElement

1. Let list be NonConstructorMethodDefinitions of ClassElementList.
2. If ClassElement is ClassElement : ; , return list.
3. If IsStatic of ClassElement is false and PropName of ClassElement is "constructor", return list.
4. Append ClassElement to the end of list.
5. Return list.

14.5.11 Static Semantics: PrototypePropertyNameList
ClassElementList : ClassElement

1. If PropName of ClassElement is empty, return a new empty List.
2. If IsStatic of ClassElement is true, return a new empty List.
3. Return a List containing PropName of ClassElement.

ClassElementList : ClassElementList ClassElement

1. Let list be PrototypePropertyNameList of ClassElementList.
2. If PropName of ClassElement is empty, return list.
3. If IsStatic of ClassElement is true, return list.
4. Append PropName of ClassElement to the end of list.
5. Return list.

14.5.12 Static Semantics: PropName
ClassElement : ;

1. Return empty.

14.5.13 Runtime Semantics: ClassDefinitionEvaluation

With parameter className.

ClassTail : ClassHeritage { ClassBody }

1. Let lex be the LexicalEnvironment of the running execution context.
2. Let classScope be NewDeclarativeEnvironment(lex).
3. Let classScopeEnvRec be classScope's EnvironmentRecord.
4. If className is not undefined, then
   a. Perform classScopeEnvRec.CreateImmutableBinding(className, true).
5. If ClassHeritageopt is not present, then
a. Let \textit{protoParent} be the intrinsic object \%ObjectPrototype\%.

b. Let \textit{constructorParent} be the intrinsic object \%FunctionPrototype\%.

6. Else,
   a. Set the \textit{running execution context}'s LexicalEnvironment to \textit{classScope}.
   b. Let \textit{superclass} be the result of evaluating \textit{ClassHeritage}.
   c. Set the \textit{running execution context}'s LexicalEnvironment to \textit{lex}.
   d. \textbf{ReturnIfAbrupt}(\textit{superclass}).
   e. If \textit{superclass} is \textbf{null}, then
      i. Let \textit{protoParent} be \textbf{null}.
      ii. Let \textit{constructorParent} be the intrinsic object \%FunctionPrototype\%.
   f. Else if \textbf{IsConstructor}(\textit{superclass}) is \textbf{false}, throw a \textbf{TypeError} exception.
   g. Else,
      i. Let \textit{protoParent} be \textbf{? \textbf{Get}(superclass, "prototype")}.
      ii. If \textbf{Type}(\textit{protoParent}) is neither Object nor \textbf{null}, throw a \textbf{TypeError} exception.
      iii. Let \textit{constructorParent} be \textit{superclass}.

7. Let \textit{proto} be \textbf{ObjectCreate}(\textit{protoParent}).
8. If \textit{ClassBody} is not present, let \textit{constructor} be \textbf{empty}.
9. Else, let \textit{constructor} be ConstructorMethod of \textit{ClassBody}.

10. If \textit{constructor} is \textbf{empty}, then
    a. If \textit{ClassHeritage} is present, then
        i. Set \textit{constructor} to the result of parsing the source text

        \textbf{constructor}(... \textbf{args})\{}
        \textbf{super}(...\textbf{args});\}

        \textbf{constructor}(\textbf{ })\{}

        using the syntactic grammar with the \textit{goal symbol} \texttt{MethodDefinition}\texttt{[~Yield, ~Await]}
   b. Else,
        i. Set \textit{constructor} to the result of parsing the source text

        \textbf{constructor}(\textbf{ })\{}

        using the syntactic grammar with the \textit{goal symbol} \texttt{MethodDefinition}\texttt{[~Yield, ~Await]}

11. Set the \textit{running execution context}'s LexicalEnvironment to \textit{classScope}.
12. Let \textit{constructorInfo} be the result of performing DefineMethod for \textit{constructor} with arguments \textit{proto} and \textit{constructorParent} as the optional \textit{functionPrototype} argument.
13. \textbf{Assert}: \textit{constructorInfo} is not an \textit{abrupt completion}.
14. Let \textit{F} be \textit{constructorInfo}.\texttt{[[Closure]]}.
15. If \textit{ClassHeritage} is present, set \textit{F}.\texttt{[[ConstructorKind]]} to \texttt{"derived"}.
16. Perform \textbf{MakeConstructor}(\textit{F, false, proto}).
17. Perform \textbf{MakeClassConstructor}(\textit{F}).
18. Perform \textbf{CreateMethodProperty}(\textit{proto, "constructor", F}).
19. If \textit{ClassBody} is not present, let \textit{methods} be a new empty \textbf{List}. 
20. Else, let `methods` be NonConstructorMethodDefinitions of `ClassBody`.
21. For each `ClassElement m` in order from `methods`, do
   a. If IsStatic of `m` is `false`, then
      i. Let `status` be the result of performing PropertyDefinitionEvaluation for `m` with arguments `proto` and `false`.
   b. Else,
      i. Let `status` be the result of performing PropertyDefinitionEvaluation for `m` with arguments `F` and `false`.
   c. If `status` is an abrupt completion, then
      i. Set the running execution context's LexicalEnvironment to `lex`.
      ii. Return `Completion(status)`.
22. Set the running execution context's LexicalEnvironment to `lex`.
23. If `className` is not `undefined`, then
   a. Perform `classScopeEnvRec.InitializeBinding(className, F)`.

### 14.5.14 Runtime Semantics: BindingClassDeclarationEvaluation

**ClassDeclaration : class BindingIdentifier ClassTail**

1. Let `className` be `StringValue` of `BindingIdentifier`.
2. Let `value` be the result of ClassDefinitionEvaluation of `ClassTail` with argument `className`.
3. ReturnIfAbrupt(`value`).
4. Let `hasOwnProperty` be `? HasOwnProperty(value, "name")`.
5. If `hasOwnProperty` is `false`, perform `SetFunctionName(value, className)`.
6. Let `env` be the running execution context's LexicalEnvironment.
7. Perform `? InitializeBoundName(className, value, env)`.
8. Return `value`.

**ClassDeclaration : class ClassTail**

1. Return the result of ClassDefinitionEvaluation of `ClassTail` with argument `undefined`.

### NOTE

**ClassDeclaration : class ClassTail** only occurs as part of an `ExportDeclaration` and the setting of a name property and establishing its binding are handled as part of the evaluation action for that production. See 15.2.3.11.

### 14.5.15 Runtime Semantics: Evaluation

**ClassDeclaration : class BindingIdentifier ClassTail**

1. Let `status` be the result of BindingClassDeclarationEvaluation of this `ClassDeclaration`.
2. ReturnIfAbrupt(`status`).
3. Return `NormalCompletion(empty)`.

**NOTE 1**

`ClassDeclaration : class ClassTail` only occurs as part of an `ExportDeclaration` and is never directly evaluated.

**ClassExpression : class BindingIdentifier ClassTail**

1. If `BindingIdentifier_opt` is not present, let `className` be `undefined`.
2. Else, let `className` be `StringValue` of `BindingIdentifier`.
3. Let `value` be the result of `ClassDefinitionEvaluation` of `ClassTail` with argument `className`.
4. Return `IfAbrupt(value)`.
5. If `className` is not `undefined`, then
   a. Let `hasOwnProperty` be `HasOwnProperty(value, "name")`.
   b. If `hasOwnProperty` is `false`, then
      i. Perform `SetFunctionName(value, className)`.
6. Return `NormalCompletion(value)`.

**NOTE 2**

If the class definition included a `name` static method then that method is not over-written with a `name` data property for the class name.

### 14.6 Async Function Definitions

**Syntax**

AsyncFunctionDeclaration[

```
    FormalParameters[~Yield, ?Await] ) { AsyncFunctionBody }
```

] + Default

AsyncFunctionExpression:

```
async [ no LineTerminator here ] function ( FormalParameters[~Yield, +Await] ) { AsyncFunctionBody }
```

AsyncMethod[

```
async [ no LineTerminator here ] PropertyName[?Yield, ?Await] (  
    UniqueFormalParameters[~Yield, +Await] ) { AsyncFunctionBody }
```

] + Default

AsyncFunctionBody:
FunctionBody[~Yield, +Await]

AwaitExpression[Yield] :

    await UnaryExpression[?Yield, +Await]

**NOTE 1**

await is parsed as an *AwaitExpression* when the [Await] parameter is present. The [Await] parameter is present in the following contexts:

- In an *AsyncFunctionBody*.
- In the *FormalParameters* of an *AsyncFunctionDeclaration* and *AsyncFunctionExpression*. *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* or *AsyncFunctionExpression*.
- In the *BindingIdentifier* of a *FunctionExpression* or *GeneratorExpression*.

**NOTE 2**

Unlike *YieldExpression*, it is a Syntax Error to omit the operand of an *AwaitExpression*. You must await something.

14.6.1 Static Semantics: Early Errors

*AsyncMethod* : async [no LineTerminator here] *PropertyName* ( *UniqueFormalParameters* ) { *AsyncFunctionBody* }

- It is a Syntax Error if *ContainsUseStrict* 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 [no LineTerminator here] function *BindingIdentifier* ( *FormalParameters* ) { *AsyncFunctionBody* }

*AsyncFunctionDeclaration* : async [no LineTerminator here] function ( *FormalParameters* ) { *AsyncFunctionBody* }
AsyncFunctionExpression : async [no LineTerminator here] function ( FormalParameters ) {
AsyncFunctionBody }

AsyncFunctionExpression : async [no LineTerminator here] function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }

- It is a Syntax Error if ContainsUseStrict 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 this production is strict code, the Early Error rules for UniqueFormalParameters : FormalParameters are applied.
- If the source code matching this production is strict code, it is a Syntax Error if BindingIdentifier is the IdentifierName eval or the IdentifierName arguments.
- It is a Syntax Error if any element of the BoundNames of FormalParameters also occurs in the LexicallyDeclaredNames of 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.

14.6.2 Static Semantics: BoundNames
AsyncFunctionDeclaration : async [no LineTerminator here] function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }

1. Return the BoundNames of BindingIdentifier.

AsyncFunctionDeclaration : async [no LineTerminator here] function ( FormalParameters ) { AsyncFunctionBody }

1. Return "*default*".

**NOTE** "*default*" is used within this specification as a synthetic name for hoistable anonymous functions that are defined using export declarations.

14.6.3 Static Semantics: ComputedPropertyContains

With parameter symbol.

AsyncMethod : async [no LineTerminator here] PropertyName ( UniqueFormalParameters ) { AsyncFunctionBody }

1. Return the result of ComputedPropertyContains for PropertyName with argument symbol.
14.6.4 Static Semantics: Contains

With parameter \textit{symbol}.

\textit{AsyncFunctionDeclaration}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} \textit{BindingIdentifier} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}
\textit{AsyncFunctionDeclaration}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}
\textit{AsyncFunctionExpression}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}
\textit{AsyncFunctionExpression}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} \textit{BindingIdentifier} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}

1. Return \texttt{false}.

14.6.5 Static Semantics: HasDirectSuper

\textit{AsyncMethod}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \textit{PropertyName} (\textit{UniqueFormalParameters} ) \{ \textit{AsyncFunctionBody} \}

1. If \textit{UniqueFormalParameters} Contains \textit{SuperCall} is \texttt{true}, return \texttt{true}.
2. Return \textit{AsyncFunctionBody} Contains \textit{SuperCall}.

14.6.6 Static Semantics: HasName

\textit{AsyncFunctionExpression}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}

1. Return \texttt{false}.

\textit{AsyncFunctionExpression}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} \textit{BindingIdentifier} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}

1. Return \texttt{true}.

14.6.7 Static Semantics: IsConstantDeclaration

\textit{AsyncFunctionDeclaration}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} \textit{BindingIdentifier} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}
\textit{AsyncFunctionDeclaration}: \texttt{async} \nobreakdash \texttt{[no LineTerminator here]} \texttt{function} (\textit{FormalParameters} ) \{ \textit{AsyncFunctionBody} \}

1. Return \texttt{false}. 
14.6.8 Static Semantics: IsFunctionDefinition
AsyncFunctionExpression : async [no LineTerminator here] function ( FormalParameters ) {
AsyncFunctionBody
}

AsyncFunctionExpression : async [no LineTerminator here] function BindingIdentifier ( FormalParameters ) {
AsyncFunctionBody
}

1. Return true.

14.6.9 Static Semantics: PropName
AsyncMethod : async [no LineTerminator here] PropertyName ( UniqueFormalParameters ) {
AsyncFunctionBody
}

1. Return PropName of PropertyName.

14.6.10 Runtime Semantics: InstantiateFunctionObject

With parameter scope.

AsyncFunctionDeclaration : async [no LineTerminator here] function BindingIdentifier ( FormalParameters ) {
AsyncFunctionBody
}

1. If the function code for AsyncFunctionDeclaration is strict mode code, let strict be true. Otherwise, let
strict be false.
2. Let name be StringValue of BindingIdentifier.
4. Perform ! SetFunctionName(F, name).
5. Return F.

AsyncFunctionDeclaration : async [no LineTerminator here] function ( FormalParameters ) {
AsyncFunctionBody
}

1. If the function code for AsyncFunctionDeclaration is strict mode code, let strict be true. Otherwise, let
strict be false.
2. Let F be ! AsyncFunctionCreate(Normal, FormalParameters, AsyncFunctionBody, scope, strict).
3. Perform ! SetFunctionName(F, "default").
4. Return F.

14.6.11 Runtime Semantics: EvaluateBody

With parameters functionObject and List argumentsList.

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 `declResult` is an abrupt completion,
   a. Perform `!Call(promiseCapability.][[Reject]], undefined, «declResult.][[Value]]»).
5. Return `Completion{[[Type]]: return, [[Value]]: promiseCapability.][Promise], [[Target]]: empty}.

14.6.12 Runtime Semantics: PropertyDefinitionEvaluation

With parameters `object` and `enumerable`.

```
AsyncMethod : async [no LineTerminator here] PropertyName ( UniqueFormalParameters ) { 
  AsyncFunctionBody  }
```

1. Let `propKey` be the result of evaluating `PropertyName`.
2. ReturnIfAbrupt(propKey).
3. If the function code for this `AsyncMethod` is strict mode code, let `strict` be `true`. Otherwise let `strict` be `false`.
4. Let `scope` be the LexicalEnvironment of the running execution context.
5. Let `closure` be `!AsyncFunctionCreate(Method, UniqueFormalParameters, AsyncFunctionBody, scope, strict)`.
6. Perform `!MakeMethod(closure, object)`.
7. Perform `!SetFunctionName(closure, propKey)`.
8. Let `desc` be the PropertyDescriptor{[[Value]]: `closure`, [[Writable]]: `true`, [[Enumerable]]: `enumerable`, [[Configurable]]: `true`}.
9. Return `? DefinePropertyOrThrow(object, propKey, desc)`.

14.6.13 Runtime Semantics: Evaluation

```
AsyncFunctionDeclaration : async [no LineTerminator here] function BindingIdentifier ( 
  FormalParameters  ) {  AsyncFunctionBody  }
```

1. Return `NormalCompletion(empty)`.

```
AsyncFunctionDeclaration : async [no LineTerminator here] function ( FormalParameters  ) {  
  AsyncFunctionBody  }
```

1. Return `NormalCompletion(empty)`.

```
AsyncFunctionExpression : async [no LineTerminator here] function ( FormalParameters  ) {  
  AsyncFunctionBody  }
```
1. If the function code for AsyncFunctionExpression is strict mode code, let strict be true. Otherwise let strict be false.
2. Let scope be the LexicalEnvironment of the running execution context.
4. Return closure.

AsyncFunctionExpression : async [no LineTerminator here] function BindingIdentifier ( FormalParameters ) { AsyncFunctionBody }

1. If the function code for AsyncFunctionExpression is strict mode code, let strict be true. Otherwise let strict be false.
2. Let scope be the LexicalEnvironment of the running execution context.
3. Let envRec be ! NewDeclarativeEnvironment(scope).
4. Let envRec be funcEnv's EnvironmentRecord.
5. Let name be StringValue of BindingIdentifier.
7. Let closure be ! AsyncFunctionCreate(Normal, FormalParameters, AsyncFunctionBody, funcEnv, strict).
8. Perform ! SetFunctionName(closure, name).

AwaitExpression : await UnaryExpression

1. Let exprRef be the result of evaluating UnaryExpression.
2. Let value be ? GetValue(exprRef).

14.7 Async Arrow Function Definitions

Syntax

AsyncArrowFunction[In, Yield, Await] :
  CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await] [no LineTerminator here] =>
  AsyncConciseBody[?In]

AsyncConciseBody[In] :
  [lookahead ≠ {}] AssignmentExpression[?In, ~Yield, +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 [no Line Terminator here] => AsyncConciseBody
the interpretation of CoverCallExpressionAndAsyncArrowHead is refined using the following grammar:

AsyncArrowHead :

14.7.1 Static Semantics: Early Errors

- It is a Syntax Error if any element of the BoundNames of AsyncArrowBindingIdentifier also occurs in the LexicallyDeclaredNames of AsyncConciseBody.

AsyncArrowFunction : CoverCallExpressionAndAsyncArrowHead [no Line Terminator here] => AsyncConciseBody

- It is a Syntax Error if CoverCallExpressionAndAsyncArrowHead Contains YieldExpression is true.
- It is a Syntax Error if CoverCallExpressionAndAsyncArrowHead Contains AwaitExpression is true.
- It is a Syntax Error if the lexical token sequence matched by CoverCallExpressionAndAsyncArrowHead cannot be parsed with no tokens left over using AsyncArrowHead as the goal symbol.
- 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 ContainsUseStrict 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.

14.7.2 Static Semantics: CoveredAsyncArrowHead
CoverCallExpressionAndAsyncArrowHead : MemberExpression Arguments

1. Return the result of parsing the lexical token stream matched by
  CoverCallExpressionAndAsyncArrowHead using AsyncArrowHead as the goal symbol.
14.7.3 Static Semantics: BoundNames
CoverCallExpressionAndAsyncArrowHead : MemberExpression Arguments

1. Let head be CoveredAsyncArrowHead of CoverCallExpressionAndAsyncArrowHead.
2. Return the BoundNames of head.

14.7.4 Static Semantics: Contains

With parameter symbol.


1. If symbol is not one of NewTarget, SuperProperty, SuperCall, super, or this, return false.
2. Return AsyncConciseBody Contains symbol.

AsyncArrowFunction : CoverCallExpressionAndAsyncArrowHead [no LineTerminator here] => 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
Normally, Contains does not look inside most function forms. However, Contains is used to detect new.target, this, and super usage within an AsyncArrowFunction.

14.7.5 Static Semantics: ContainsExpression
AsyncArrowBindingIdentifier : BindingIdentifier

1. Return false.

14.7.6 Static Semantics: ExpectedArgumentCount
AsyncArrowBindingIdentifier : BindingIdentifier

1. Return 1.

14.7.7 Static Semantics: HasName
AsyncArrowFunction : CoverCallExpressionAndAsyncArrowHead [no LineTerminator here] =>
AsyncConciseBody

1. Return false.

14.7.8 Static Semantics: IsSimpleParameterList
AsyncArrowBindingIdentifier : BindingIdentifier

1. Return true.

CoverCallExpressionAndAsyncArrowHead : MemberExpression Arguments

1. Let head be CoveredAsyncArrowHead of CoverCallExpressionAndAsyncArrowHead.
2. Return IsSimpleParameterList of head.

14.7.9 Static Semantics: LexicallyDeclaredNames
AsyncConciseBody : [lookahead ≠ {}] AssignmentExpression

1. Return a new empty List.

14.7.10 Static Semantics: LexicallyScopedDeclarations
AsyncConciseBody : [lookahead ≠ {}] AssignmentExpression

1. Return a new empty List.

14.7.11 Static Semantics: VarDeclaredNames
AsyncConciseBody : [lookahead ≠ {}] AssignmentExpression

1. Return a new empty List.

14.7.12 Static Semantics: VarScopedDeclarations
AsyncConciseBody : [lookahead ≠ {}] AssignmentExpression

1. Return a new empty List.

14.7.13 Runtime Semantics: IteratorBindingInitialization

With parameters iteratorRecord and environment.

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

### 14.7.14 Runtime Semantics: EvaluateBody

With parameters `functionObject` and `List argumentsList`.

**AsyncConciseBody** : `[lookahead ≠ {}] AssignmentExpression

1. Let `promiseCapability` be `! NewPromiseCapability(%Promise%)`.
2. Let `declResult` be `! FunctionDeclarationInstantiation(functionObject, argumentsList)`.
3. If `declResult` is not an abrupt completion, then
   a. Perform `! AsyncFunctionStart(promiseCapability, AssignmentExpression)`.
4. Else `declResult` is an abrupt completion,
   a. Perform `! Call(promiseCapability.[[Reject]], undefined, «declResult.[[Value]]»)`.
5. Return `Completion{[[Type]]: return, [[Value]]: promiseCapability.[[Promise]], [[Target]]: empty}`.

**AsyncConciseBody** : `{ AsyncFunctionBody }`

1. Return the result of EvaluateBody of `AsyncFunctionBody` passing `functionObject` and `argumentsList` as the arguments.

### 14.7.15 Runtime Semantics: Evaluation


1. If the function code for this `AsyncArrowFunction` is strict mode code, let `strict` be **true**. Otherwise, let `strict` be **false**.
2. Let `scope` be the LexicalEnvironment of the running execution context.
3. Let `parameters` be `AsyncArrowBindingIdentifier`.
4. Let `closure` be `! AsyncFunctionCreate(Arrow, parameters, AsyncConciseBody, scope, strict)`.
AsyncArrowFunction : CoverCallExpressionAndAsyncArrowHead =>
AsyncConciseBody

1. If the function code for this AsyncArrowFunction is strict mode code, let strict be true. Otherwise, let strict be false.
2. Let scope be the LexicalEnvironment of the running execution context.
3. Let head be CoveredAsyncArrowHead of CoverCallExpressionAndAsyncArrowHead.
4. Let parameters be the ArrowFormalParameters of head.
5. Let closure be ! AsyncFunctionCreate(Arrow, parameters, AsyncConciseBody, scope, strict).

14.8 Tail Position Calls

14.8.1 Static Semantics: IsInTailPosition( call )

The abstract operation IsInTailPosition with argument call performs the following steps:

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 an AsyncConciseBody, return false.
8. 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 9.2.7) that enables observation of the chain of caller contexts.

14.8.2 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.

14.8.2.1 Statement Rules
ConciseBody : AssignmentExpression

1. Return HasCallInTailPosition of AssignmentExpression with argument call.

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

IterationStatement :
  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

IterationStatement :
  do Statement while ( Expression );
while ( Expression ) Statement
for ( Expression_opt ; Expression_opt ; Expression_opt ) Statement
for ( var VariableDeclarationList ; Expression_opt ; Expression_opt ) Statement
for ( LexicalDeclaration Expression_opt ; Expression_opt ) Statement
for ( LeftHandSideExpression in Expression ) Statement
for ( var ForBinding in Expression ) Statement
for ( ForDeclaration in Expression ) 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 DefaultClause CaseClauses }

  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 the 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
**DefaultClause**: `default : StatementList`

1. If `StatementList` is present, return `HasCallInTailPosition of StatementList with argument call`.
2. Return `false`.

**TryStatement**: `try Block Catch`

1. Return `HasCallInTailPosition of Catch with argument call`.

**TryStatement**: `try Block Finally`

**TryStatement**: `try Block Catch Finally`

1. Return `HasCallInTailPosition of Finally with argument call`.

**Catch**: `catch ( CatchParameter ) Block`

1. Return `HasCallInTailPosition of Block with argument call`.

### 14.8.2.2 Expression Rules

**NOTE** A potential tail position call that is immediately followed by return `GetValue` of the call result is also a possible tail position call. Function calls cannot return reference values, so such a `GetValue` operation will always return the same value as the actual function call result.

**AssignmentExpression**:  
- `YieldExpression`
- `ArrowFunction`
- `AsyncArrowFunction`
- `LeftHandSideExpression = AssignmentExpression`
- `LeftHandSideExpression AssignmentOperator AssignmentExpression`

**BitwiseANDExpression**: `BitwiseANDExpression & EqualityExpression`

**BitwiseXORExpression**: `BitwiseXORExpression ^ BitwiseANDExpression`

**BitwiseORExpression**: `BitwiseORExpression | BitwiseXORExpression`

**EqualityExpression**:  
- `EqualityExpression == RelationalExpression`
- `EqualityExpression != RelationalExpression`
- `EqualityExpression === RelationalExpression`
- `EqualityExpression !== RelationalExpression`

**RelationalExpression**:  
- `RelationalExpression < ShiftExpression`
RelationalExpression \rightarrow ShiftExpression
RelationalExpression \leq ShiftExpression
RelationalExpression \geq ShiftExpression
RelationalExpression instanceof ShiftExpression
RelationalExpression in ShiftExpression

ShiftExpression :
  ShiftExpression \ll AdditiveExpression
  ShiftExpression \gg AdditiveExpression
  ShiftExpression \ggg 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
  RegularExpressionLiteral
  TemplateLiteral

1. Return false.

Expression :
  AssignmentExpression
  Expression , AssignmentExpression

1. Return HasCallInTailPosition of AssignmentExpression with argument call.

ConditionalExpression : LogicalORExpression ? 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.

CallExpression :
  CoverCallExpressionAndAsyncArrowHead
1. If this CallExpression is call, return true.
2. Return false.

MemberExpression :
   CallExpression 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.

14.8.3 Runtime Semantics: PrepareForTailCall ()

The abstract operation PrepareForTailCall performs the following steps:

1. Let leafContext be the running execution context.
2. Suspend leafContext.
3. Pop leafContext from the execution context 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.
15 ECMAScript Language: Scripts and Modules

15.1 Scripts

Syntax

\[\text{Script} : \]
\[\text{ScriptBody}_{\text{opt}}\]

\[\text{ScriptBody} : \]
\[\text{StatementList}[^{\text{Yield, Await, Return}}]\]

15.1.1 Static Semantics: Early Errors

\textit{Script : ScriptBody}

- It is a Syntax Error if the LexicallyDeclaredNames of \textit{ScriptBody} contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of \textit{ScriptBody} also occurs in the VarDeclaredNames of \textit{ScriptBody}.

\textit{ScriptBody : StatementList}

- It is a Syntax Error if \textit{StatementList} contains \texttt{super} unless the source code containing \texttt{super} is eval code that is being processed by a \texttt{direct eval}. Additional early error rules for \texttt{super} within \texttt{direct eval} are defined in 18.2.1.1.
- It is a Syntax Error if \textit{StatementList} contains \texttt{NewTarget} unless the source code containing \texttt{NewTarget} is eval code that is being processed by a \texttt{direct eval}. Additional early error rules for \texttt{NewTarget} in \texttt{direct eval} are defined in 18.2.1.1.
- It is a Syntax Error if \text{ContainsDuplicateLabels of StatementList with argument « »} is \texttt{true}.
- It is a Syntax Error if \text{ContainsUndefinedBreakTarget of StatementList with argument « »} is \texttt{true}.
- It is a Syntax Error if \text{ContainsUndefinedContinueTarget of StatementList with arguments « » and « »} is \texttt{true}.

15.1.2 Static Semantics: IsStrict

\textit{ScriptBody : StatementList}

1. If the \textit{Directive Prologue} of \textit{StatementList} contains a \texttt{Use Strict Directive}, return \texttt{true}; otherwise, return \texttt{false}.

15.1.3 Static Semantics: LexicallyDeclaredNames
1. Return TopLevelLexicallyDeclaredNames of \textit{StatementList}.

\textbf{NOTE} At the top level of a \textit{Script}, function declarations are treated like var declarations rather than like lexical declarations.

\subsection*{15.1.4 Static Semantics: LexicallyScopedDeclarations}
\textit{ScriptBody} : \textit{StatementList}

1. Return TopLevelLexicallyScopedDeclarations of \textit{StatementList}.

\subsection*{15.1.5 Static Semantics: VarDeclaredNames}
\textit{ScriptBody} : \textit{StatementList}

1. Return TopLevelVarDeclaredNames of \textit{StatementList}.

\subsection*{15.1.6 Static Semantics: VarScopedDeclarations}
\textit{ScriptBody} : \textit{StatementList}

1. Return TopLevelVarScopedDeclarations of \textit{StatementList}.

\subsection*{15.1.7 Runtime Semantics: Evaluation}
\textit{Script} : [empty]

1. Return \texttt{NormalCompletion(undefined)}.

\subsection*{15.1.8 Script Records}

A \textit{Script Record} encapsulates information about a script being evaluated. Each script record contains the fields listed in \textit{Table 36}.

\begin{table}[h]
\centering
\caption{\textit{Script Record} Fields}
\begin{tabular}{|l|l|l|}
\hline
Field Name & Value Type & Meaning \\
\hline
[[Realm]] & Realm Record | undefined & The realm within which this script was created. \texttt{undefined} if not yet assigned. \\
\hline
[[Environment]] & Lexical Environment | undefined & The Lexical Environment containing the top level bindings for this script. This field is set when the script is instantiated. \\
\hline
\end{tabular}
\end{table}
<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 Script as the goal symbol.</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td>Any, default value is undefined.</td>
<td>Field reserved for use by host environments that need to associate additional information with a script.</td>
</tr>
</tbody>
</table>

### 15.1.9 ParseScript (sourceText, realm, hostDefined)

The abstract operation ParseScript with arguments sourceText, realm, and hostDefined creates a Script Record based upon the result of parsing sourceText as a Script. ParseScript performs the following steps:

1. Assert: sourceText is an ECMAScript source text (see clause 10).
2. Parse sourceText using Script as the goal symbol and analyse the parse result for any Early Error conditions.
   - If the parse was successful and no early errors were found, let body be the resulting parse tree.
   - Otherwise, let body be a List of one or more SyntaxError or ReferenceError objects representing the parsing errors and/or early errors.
   - Parsing and early error detection may be interwoven in an implementation-dependent manner. If more than one parsing error or early error is present, the number and ordering of error objects in the list is implementation-dependent, but at least one must be present.
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.

### 15.1.10 ScriptEvaluation (scriptRecord)

1. Let globalEnv be scriptRecord.[[Realm]].[[GlobalEnv]].
2. Let scriptCxt be a new ECMAScript code execution context.
3. Set the Function of scriptCxt to null.
4. Set the Realm of scriptCxt to scriptRecord.[[Realm]].
5. Set the ScriptOrModule of scriptCxt to scriptRecord.
6. Set the VariableEnvironment of scriptCxt to globalEnv.
7. Set the LexicalEnvironment of scriptCxt to globalEnv.
8. Suspend the currently running execution context.
9. Push `scriptCxt` on to the execution context stack; `scriptCxt` 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 `scriptCxt` and remove it from the execution context stack.
15. Assert: The execution context stack is not empty.
16. Resume the context that is now on the top of the execution context stack as the running execution context.
17. Return `Completion(result)`.

### 15.1.11 Runtime Semantics: GlobalDeclarationInstantiation (script, env)

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

GlobalDeclarationInstantiation is performed as follows using arguments `script` and `env`. `script` is the `ScriptBody` for which the execution context is being established. `env` is the global lexical environment in which bindings are to be created.

1. Let `envRec` be `env`'s EnvironmentRecord.
2. Assert: `envRec` is a global Environment Record.
3. Let `lexNames` be the LexicallyDeclaredNames of `script`.
4. Let `varNames` be the VarDeclaredNames of `script`.
5. For each `name` in `lexNames`, do
   a. If `envRec.HasVarDeclaration(name)` is `true`, throw a SyntaxError exception.
   b. If `envRec.HasLexicalDeclaration(name)` is `true`, throw a SyntaxError exception.
   c. Let `hasRestrictedGlobal` be `envRec.HasRestrictedGlobalProperty(name)`.
   d. If `hasRestrictedGlobal` is `true`, throw a SyntaxError exception.
6. For each `name` in `varNames`, do
   a. If `envRec.HasLexicalDeclaration(name)` is `true`, throw a SyntaxError exception.
7. Let `varDeclarations` be the VarScopedDeclarations of `script`.
8. Let `functionsToInitialize` be a new empty List.
9. Let `declaredFunctionNames` be a new empty List.
10. For each `d` in `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`, or an `AsyncFunctionDeclaration`.

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`.

11. Let `declaredVarNames` be a new empty `List`.

12. For each $d$ in `varDeclarations`, do
   a. If $d$ is a `VariableDeclaration`, a `ForBinding`, or a `BindingIdentifier`, then
      i. For each String $vn$ in 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`.

13. 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.

14. NOTE: Annex B.3.3.2 adds additional steps at this point.

15. Let `lexDeclarations` be the LexicallyScopedDeclarations of `script`.

16. For each element $d$ in `lexDeclarations`, do
   a. NOTE: Lexically declared names are only instantiated here but not initialized.
   b. For each element $dn$ of the BoundNames of $d$, do
      i. If IsConstantDeclaration of $d$ is `true`, then
      ii. Else,

17. For each Parse Node $f$ in `functionsToInitialize`, do
   a. Let $fn$ be the sole element of the BoundNames of $f$.
   b. Let $fo$ be the result of performing InstantiateFunctionObject for $f$ with argument $env$.

18. For each String $vn$ in `declaredVarNames`, in list order, do

19. Return `NormalCompletion`(empty).
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.

15.1.12 Runtime Semantics: ScriptEvaluationJob (sourceText, hostDefined)

The job ScriptEvaluationJob with parameters sourceText and hostDefined parses, validates, and evaluates sourceText as a Script.

1. Assert: sourceText is an ECMAScript source text (see clause 10).
2. Let realm be the current Realm Record.
3. Let s be ParseScript(sourceText, realm, hostDefined).
4. If s is a List of errors, then
   a. Perform HostReportErrors(s).
   b. Return NormalCompletion(undefined).
5. Return ? ScriptEvaluation(s).

15.2 Modules

Syntax

Module : ModuleBody opt

ModuleBody : ModuleItemList

ModuleItemList : ModuleItem ModuleItemList ModuleItem

ModuleItem : ImportDeclaration
15.2.1 Module Semantics

15.2.1.1 Static Semantics: Early Errors

*ModuleBody : ModuleItemList*

- It is a Syntax Error if the LexicallyDeclaredNames of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *ModuleItemList* also occurs in the VarDeclaredNames of *ModuleItemList*.
- It is a Syntax Error if the ExportedNames of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if any element of the ExportedBindings of *ModuleItemList* does not also occur in either the VarDeclaredNames of *ModuleItemList*, or the LexicallyDeclaredNames of *ModuleItemList*.
- It is a Syntax Error if *ModuleItemList* Contains `super`.
- It is a Syntax Error if *ModuleItemList* Contains `NewTarget`.
- It is a Syntax Error if ContainsDuplicateLabels of *ModuleItemList* with argument « » is `true`.
- It is a Syntax Error if ContainsUndefinedBreakTarget of *ModuleItemList* with argument « » is `true`.
- It is a Syntax Error if ContainsUndefinedContinueTarget of *ModuleItemList* with arguments « » and « » is `true`.

**NOTE**
The duplicate ExportedNames rule implies that multiple `export default` *ExportDeclaration* items within a *ModuleBody* is a Syntax Error. Additional error conditions relating to conflicting or duplicate declarations are checked during module linking prior to evaluation of a *Module*. If any such errors are detected the *Module* is not evaluated.

15.2.1.2 Static Semantics: ContainsDuplicateLabels

With parameter `labelSet`.

*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`.
15.2.1.3 Static Semantics: ContainsUndefinedBreakTarget

With parameter \textit{labelSet}.

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

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

\textit{ModuleItem} :
\begin{itemize}
  \item ImportDeclaration
  \item ExportDeclaration
\end{itemize}

1. Return \texttt{false}.

15.2.1.4 Static Semantics: ContainsUndefinedContinueTarget

With parameters \textit{iterationSet} and \textit{labelSet}.

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

1. Let \textit{hasUndefinedLabels} be ContainsUndefinedContinueTarget of \textit{ModuleItemList} with arguments \textit{iterationSet} and « ».
2. If \textit{hasUndefinedLabels} is \texttt{true}, return \texttt{true}.
3. Return ContainsUndefinedContinueTarget of \textit{ModuleItem} with arguments \textit{iterationSet} and « ».

\textit{ModuleItem} :
\begin{itemize}
  \item ImportDeclaration
  \item ExportDeclaration
\end{itemize}

1. Return \texttt{false}.

15.2.1.5 Static Semantics: ExportedBindings

\begin{itemize}
  \item \textbf{NOTE} ExportedBindings are the locally bound names that are explicitly associated with a \textit{Module} 's ExportedNames.
\end{itemize}

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

1. Let \textit{names} be ExportedBindings of \textit{ModuleItemList}.
2. Append to \textit{names} the elements of the ExportedBindings of \textit{ModuleItem}.
3. Return \textit{names}.

\textit{ModuleItem} :
1. Return a new empty List.

### 15.2.1.6 Static Semantics: ExportedNames

**NOTE** ExportedNames are the externally visible names that a Module explicitly maps to one of its local name bindings.

\[
\text{ModuleItemList} : \text{ModuleItemList ModuleItem}
\]

1. Let \( \text{names} \) be ExportedNames of \( \text{ModuleItemList} \).
2. Append to \( \text{names} \) the elements of the ExportedNames of \( \text{ModuleItem} \).
3. Return \( \text{names} \).

\[
\text{ModuleItem} : \text{ExportDeclaration}
\]

1. Return the ExportedNames of \( \text{ExportDeclaration} \).

\[
\text{ModuleItem} : \text{ImportDeclaration} \\
\text{StatementListItem}
\]

1. Return a new empty List.

### 15.2.1.7 Static Semantics: ExportEntries

\[
\text{Module} : [\text{empty}]
\]

1. Return a new empty List.

\[
\text{ModuleItemList} : \text{ModuleItemList ModuleItem}
\]

1. Let \( \text{entries} \) be ExportEntries of \( \text{ModuleItemList} \).
2. Append to \( \text{entries} \) the elements of the ExportEntries of \( \text{ModuleItem} \).
3. Return \( \text{entries} \).

\[
\text{ModuleItem} : \text{ImportDeclaration} \\
\text{StatementListItem}
\]

1. Return a new empty List.

### 15.2.1.8 Static Semantics: ImportEntries
Module : [empty]

1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem

1. Let entries be ImportEntries of ModuleItemList.
2. Append to entries the elements of the ImportEntries of ModuleItem.
3. Return entries.

ModuleItem :

ExportDeclaration
StatementListItem

1. Return a new empty List.

15.2.1.9 Static Semantics: ImportedLocalNames (importEntries)

The abstract operation ImportedLocalNames with argument importEntries creates a List of all of the local name bindings defined by a List of ImportEntry Records (see Table 40). ImportedLocalNames performs the following steps:

1. Let localNames be a new empty List.
2. For each ImportEntry Record i in importEntries, do
   a. Append i.[[LocalName]] to localNames.
3. Return localNames.

15.2.1.10 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.

15.2.1.11 Static Semantics: LexicallyDeclaredNames

NOTE 1 The LexicallyDeclaredNames of a Module includes the names of all of its imported bindings.

ModuleItemList : ModuleItemList ModuleItem

1. Let names be LexicallyDeclaredNames of ModuleItemList.
2. Append to names the elements of the LexicallyDeclaredNames of ModuleItem.
3. Return names.

ModuleItem : ImportDeclaration

1. Return the BoundNames of ImportDeclaration.

ModuleItem : ExportDeclaration

1. If ExportDeclaration is export VariableStatement, return a new empty List.
2. Return the BoundNames of ExportDeclaration.

ModuleItem : StatementListItem

1. Return LexicallyDeclaredNames of StatementListItem.

NOTE 2 At the top level of a Module, function declarations are treated like lexical declarations rather than like var declarations.

15.2.1.12 Static Semantics: LexicallyScopedDeclarations

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.
15.2.1.13 Static Semantics: VarDeclaredNames

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.

15.2.1.14 Static Semantics: VarScopedDeclarations

Module : [empty]

1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem

1. Let declarations be VarScopedDeclarations of ModuleItemList.
2. Append to declarations the elements of the VarScopedDeclarations of ModuleItem.
3. Return declarations.

ModuleItem : ImportDeclaration

1. Return a new empty List.

ModuleItem : ExportDeclaration

1. If ExportDeclaration is export VariableStatement, return VarScopedDeclarations of VariableStatement.
2. Return a new empty List.

15.2.1.15 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 concrete subclasses. This specification only defines a single Module Record 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 37. All Module Definition subclasses include at least those fields. Module Record also defines the abstract method list in Table 38. All Module definition subclasses must provide concrete implementations of these abstract methods.

### Table 37: Module Record Fields

<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><code>undefined</code></td>
</tr>
<tr>
<td>[[Environment]]</td>
<td>`Lexical Environment</td>
<td><code>undefined</code></td>
</tr>
<tr>
<td>[[Namespace]]</td>
<td>`Object</td>
<td><code>undefined</code></td>
</tr>
<tr>
<td>[[Evaluated]]</td>
<td><code>Boolean</code></td>
<td>Initially <code>false, true</code> if evaluation of this module has started. Remains <code>true</code> when evaluation completes, even if it is an abrupt completion.</td>
</tr>
<tr>
<td>[[HostDefined]]</td>
<td><code>Any, default value is undefined</code></td>
<td>Field reserved for use by host environments that need to associate additional information with a module.</td>
</tr>
</tbody>
</table>

### Table 38: 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 <code>ResolvedBinding Record</code>, of the form <code>{{[Module]}: Module Record, [[BindingName]]: String}. Return </code>null` if the name cannot be resolved, or &quot;ambiguous&quot; if multiple bindings were found.</td>
</tr>
</tbody>
</table>
ModuleDeclarationInstantiation()  
Transitively resolve all module dependencies and create a module Environment Record for the module.

ModuleEvaluation()  
Do nothing if this module has already been evaluated. Otherwise, transitively evaluate all module dependencies of this module and then evaluate this module.
ModuleDeclarationInstantiation must be completed prior to invoking this method.

15.2.1.16 Source Text Module Records

A Source Text Module Record is used to represent information about a module that was defined from ECMAScript source text (10) 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, instantiate, and evaluate the module.

In addition to the fields, defined in Table 37, Source Text Module Records have the additional fields listed in Table 39. Each of these fields initially has the value undefined.

Table 39: 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 Module as the goal symbol.</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>
<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.</td>
</tr>
</tbody>
</table>
A List of ExportEntry records derived from the code of this module that correspond to export * declarations that occur within the module.

An ImportEntry Record is a Record that digests information about a single declarative import. Each ImportEntry Record has the fields defined in Table 40:

Table 40: 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 ModuleSpecifier of the ImportDeclaration.</td>
</tr>
<tr>
<td>[[ImportName]]</td>
<td>String</td>
<td>The name under which the desired binding is exported by the module</td>
</tr>
<tr>
<td></td>
<td></td>
<td>identified by [[ModuleRequest]]. The value &quot;*&quot; indicates that the import</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the importing module.</td>
</tr>
</tbody>
</table>

NOTE 1: Table 41 gives examples of ImportEntry records fields used to represent the syntactic import forms:

Table 41 (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>import v from &quot;mod&quot;;</td>
<td>&quot;mod&quot;</td>
<td>&quot;default&quot;</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td>import * as ns from &quot;mod&quot;;</td>
<td>&quot;mod&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;ns&quot;</td>
</tr>
<tr>
<td>import {x} from &quot;mod&quot;;</td>
<td>&quot;mod&quot;</td>
<td>&quot;x&quot;</td>
<td>&quot;x&quot;</td>
</tr>
<tr>
<td>import {x as v} from &quot;mod&quot;;</td>
<td>&quot;mod&quot;</td>
<td>&quot;x&quot;</td>
<td>&quot;v&quot;</td>
</tr>
<tr>
<td>import &quot;mod&quot;;</td>
<td></td>
<td></td>
<td>An ImportEntry Record is not created.</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 42:
<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>null</td>
</tr>
<tr>
<td>[[ImportName]]</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>[[LocalName]]</td>
<td>String</td>
<td>null</td>
</tr>
</tbody>
</table>

**NOTE 2** Table 43 gives examples of the ExportEntry record fields used to represent the syntactic exp

### Table 43 (Informative): Export Forms Mappings to ExportEntry Record

<table>
<thead>
<tr>
<th>Export Statement Form</th>
<th>[[ExportName]]</th>
<th>[[ModuleRequest]]</th>
<th>[[ImportName]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>export var v;</td>
<td>&quot;v&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>export default function f(){}</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>export default function(){}</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>export default 42;</td>
<td>&quot;default&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>export {x};</td>
<td>&quot;x&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>export {v as x};</td>
<td>&quot;x&quot;</td>
<td>null</td>
<td>null</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>
</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>
</tr>
<tr>
<td>export * from &quot;mod&quot;;</td>
<td>null</td>
<td>&quot;mod&quot;</td>
<td>&quot;*&quot;</td>
</tr>
</tbody>
</table>

The following definitions specify the required concrete methods and other abstract operations for Source Text Module Records.
15.2.1.16.1 **ParseModule** *(sourceText, realm, hostDefined)*

The abstract operation ParseModule with arguments *sourceText*, *realm*, and *hostDefined* creates a *Source Text Module Record* based upon the result of parsing *sourceText* as a *Module*. ParseModule performs the following steps:

1. Assert: *sourceText* is an ECMAScript source text (see clause 10).
2. Parse *sourceText* using *Module* as the *goal symbol* and analyse the parse result for any Early Error conditions. If the parse was successful and no early errors were found, let *body* be the resulting parse tree. Otherwise, let *body* be a *List* of one or more *SyntaxError* or *ReferenceError* objects representing the parsing errors and/or early errors. Parsing and early error detection may be interwoven in an implementation-dependent manner. If more than one parsing error or early error is present, the number and ordering of error objects in the list is implementation-dependent, but at least one must be present.
3. If *body* is a *List* of errors, return *body*.
4. Let *requestedModules* be the ModuleRequests of *body*.
5. Let *importEntries* be ImportEntries of *body*.
6. Let *importedBoundNames* be *ImportedLocalNames*(*importEntries*).
7. Let *indirectExportEntries* be a new empty *List*.
8. Let *localExportEntries* be a new empty *List*.
9. Let *starExportEntries* be a new empty *List*.
10. Let *exportEntries* be ExportEntries of *body*.
11. For each *ExportEntry Record* *ee* in *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*.[[ModuleName]] is "**", then
            a. Assert: This is a re-export of an imported module namespace object.
            b. Append *ee* to *localExportEntries*.
         3. Else this is a re-export of a single name,
            a. Append the *ExportEntry Record* *[[ModuleRequest]]: ie.[[ModuleName]], [[ModuleName]]: ie.[[ModuleName]], [[LocalName]]: *null*, [[ExportName]]: *ee*, [[ExportName]]: *ee*} to *indirectExportEntries*.
      b. Else if *ee*.[[ModuleName]] is "**", then
         i. Append *ee* to *starExportEntries*.
   c. Else,
      i. Append *ee* to *indirectExportEntries*. 
12. Return Source Text Module Record 

```
{{[Realm]: realm, [Environment]: undefined, [HostDefined]: hostDefined, [Namespace]: undefined, [Evaluated]: false, [ECMAScriptCode]: body, [RequestedModules]: requestedModules, [ImportEntries]: importEntries, [LocalExportEntries]: localExportEntries, [StarExportEntries]: starExportEntries, [IndirectExportEntries]: indirectExportEntries}}.
```

*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.

### 15.2.1.16.2 GetExportedNames( *exportStarSet* ) Concrete Method

The GetExportedNames concrete method of a Source Text Module Record with argument *exportStarSet* performs the following steps:

1. Let *module* be this Source Text Module Record.
2. If *exportStarSet* contains *module*, then
   a. Assert: We’ve reached the starting point of an *import *circularity.
   b. Return a new empty List.
3. Append *module* to *exportStarSet*.
4. Let *exportedNames* be a new empty List.
5. For each ExportEntry Record *e* in *module.*[[LocalExportEntries]], do
   a. Assert: *module* provides the direct binding for this export.
   b. Append *e.*[[ExportName]] to *exportedNames*.
6. For each ExportEntry Record *e* in *module.*[[IndirectExportEntries]], do
   a. Assert: *module* imports a specific binding for this export.
   b. Append *e.*[[ExportName]] to *exportedNames*.
7. For each ExportEntry Record *e* in *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*.
8. Return *exportedNames*.

*NOTE* GetExportedNames does not filter out or throw an exception for names that have ambiguous star export bindings.

### 15.2.1.16.3 ResolveExport( *exportName, resolveSet* ) Concrete Method
The ResolveExport concrete method of a Source Text Module Record with arguments exportName, and resolveSet performs the following steps:

1. Let module be this Source Text Module Record.
2. For each Record {[[Module]], [[ExportName]]} r in resolveSet, do
   a. If module and r.[[Module]] are the same Module Record and SameValue(exportName, r. [[ExportName]]) is true, then
      i. Assert: This is a circular import request.
      ii. Return null.
3. Append the Record {[[Module]]: module, [[ExportName]]: exportName} to resolveSet.
4. For each ExportEntry Record e in 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]]}.
5. For each ExportEntry Record e in module.[[IndirectExportEntries]], do
   a. If SameValue(exportName, e.[[ExportName]]) is true, then
      i. Assert: module imports a specific binding for this export.
      ii. Let importedModule be ? HostResolveImportedModule(module, e.[[ModuleRequest]]).
      iii. Return ? importedModule.ResolveExport(e.[[ImportName]], resolveSet).
6. 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 *.
7. Let starResolution be null.
8. For each ExportEntry Record e in 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. If starResolution is null, set starResolution to resolution.
      ii. 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".
9. Return starResolution.

NOTE 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 use 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. 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.

### 15.2.1.16.4 ModuleDeclarationInstantiation() Concrete Method

The ModuleDeclarationInstantiation concrete method of a Source Text Module Record performs the following steps:

1. Let module be this Source Text Module Record.
2. Let realm be module.[[Realm]].
3. Assert: realm is not undefined.
4. Let code be module.[[ECMAScriptCode]].
5. If module.[[Environment]] is not undefined, return NormalCompletion(empty).
6. Let env be NewModuleEnvironment(realm.[[GlobalEnv]]).
7. Set module.[[Environment]] to env.
8. For each String required that is an element of module.[[RequestedModules]], do
   a. NOTE: Before instantiating a module, all of the modules it requested must be available. An implementation may perform this test at any time prior to this point.
   b. Let requiredModule be ? HostResolveImportedModule(module, required).
   c. Perform ? requiredModule.ModuleDeclarationInstantiation().
9. For each ExportEntry Record e in module.[[IndirectExportEntries]], do
   a. Let resolution be ? module.ResolveExport(e.[[ExportName]], « »).
   b. If resolution is null or resolution is "ambiguous", throw a SyntaxError exception.
10. Assert: All named exports from module are resolvable.
11. Let envRec be env's EnvironmentRecord.
12. For each ImportEntry Record in in module.[[ImportEntries]], do
   a. Let importedModule be ! HostResolveImportedModule(module, in.[[ModuleRequest]]).
   b. NOTE: The above call cannot fail because imported module requests are a subset of module.[[RequestedModules]], and these have been resolved earlier in this algorithm.
   c. If in.[[ImportName]] is "*", then
      i. Let namespace be ? GetModuleNameSpace(importedModule).
      ii. Perform ! envRec.CreateImmutableBinding(in.[[LocalName]], true).
      iii. Call envRec.InitializeBinding(in.[[LocalName]], namespace).
   d. Else,
      i. Let resolution be ? importedModule.ResolveExport(in.[[ImportName]], « »).
ii. If \textit{resolution} is \textbf{null} or \textit{resolution} is "\textbf{ambiguous}", throw a \textbf{SyntaxError} exception.

iii. Call \texttt{envRec.CreateImportBinding(\texttt{in.}[\texttt{LocalName}], \texttt{resolution.}[\texttt{Module}], \texttt{resolution.}[\texttt{[BindingName]}]).}

13. Let \textit{varDeclarations} be the VarScopedDeclarations of \textit{code}.

14. Let \textit{declaredVarNames} be a new empty \texttt{List}.

15. For each element \textit{d} in \textit{varDeclarations}, do
   a. For each element \textit{dn} of the BoundNames of \textit{d}, do
      i. If \textit{dn} is not an element of \textit{declaredVarNames}, then
         1. Perform \texttt{! envRec.CreateMutableBinding(dn, false)}.
         2. Call \texttt{envRec.InitializeBinding(dn, undefined)}.
         3. Append \textit{dn} to \textit{declaredVarNames}.

16. Let \textit{lexDeclarations} be the LexicallyScopedDeclarations of \textit{code}.

17. For each element \textit{d} in \textit{lexDeclarations}, do
   a. For each element \textit{dn} of the BoundNames of \textit{d}, do
      i. If IsConstantDeclaration of \textit{d} is \textbf{true}, then
         1. Perform \texttt{! envRec.CreateImmutableBinding(dn, true)}.
      ii. Else,
         1. Perform \texttt{! envRec.CreateMutableBinding(dn, false)}.
      iii. If \textit{d} is a \textit{FunctionDeclaration}, a \textit{GeneratorDeclaration}, or an \textit{AsyncFunctionDeclaration}, then
          1. Let \textit{fo} be the result of performing InstantiateFunctionObject for \textit{d} with argument \texttt{env}.
          2. Call \texttt{envRec.InitializeBinding(dn, fo)}.

18. Return \texttt{NormalCompletion(empty)}.

\subsection*{15.2.1.16.5 \texttt{ModuleEvaluation()} Concrete Method}

The \texttt{ModuleEvaluation} concrete method of a \textbf{Source Text Module Record} performs the following steps:

1. Let \textit{module} be this \textbf{Source Text Module Record}.

2. Assert: ModuleDeclarationInstantiation has already been invoked on \textit{module} and successfully completed.

3. Assert: \textit{module.}[[\texttt{Realm}]] is not \textbf{undefined}.

4. If \textit{module.}[[\texttt{Evaluated}]] is \textbf{true}, return \textbf{undefined}.

5. Set \textit{module.}[[\texttt{Evaluated}]] to \textbf{true}.

6. For each String \textit{required} that is an element of \textit{module.}[[\texttt{RequestedModules}]], do
   a. Let \textit{requiredModule} be \texttt{! HostResolveImportedModule(module, required)}.
   b. NOTE: ModuleDeclarationInstantiation must be completed prior to invoking this method, so every requested module is guaranteed to resolve successfully.
   c. Perform \texttt{? requiredModule.ModuleEvaluation()}

7. Let \textit{moduleCxt} be a new ECMAScript code \textbf{execution context}.
8. Set the Function of `moduleCxt` to `null`.
9. Set the `Realm` of `moduleCxt` to `module`.\[[\text{Realm}]\].
10. Set the ScriptOrModule of `moduleCxt` to `module`.
11. Assert: `module` has been linked and declarations in its `module environment` have been instantiated.
12. Set the `VariableEnvironment` of `moduleCxt` to `module`.\[[\text{Environment}]\].
13. Set the `LexicalEnvironment` of `moduleCxt` to `module`.\[[\text{Environment}]\].
14. `Suspend` the currently running execution context.
15. Push `moduleCxt` on to the execution context stack; `moduleCxt` is now the running execution context.
16. Let `result` be the result of evaluating `module`.\[[\text{ECMAScriptCode}]\].
17. `Suspend moduleCxt` and remove it from the execution context stack.
18. Resume the context that is now on the top of the execution context stack as the running execution context.
19. Return `Completion(result)`.

15.2.1.17 Runtime Semantics: `HostResolveImportedModule` (`referencingModule`, `specifier`)

`HostResolveImportedModule` is an implementation-defined abstract operation that provides the concrete `Module Record` subclass instance that corresponds to the `ModuleSpecifier` String, `specifier`, occurring within the context of the module represented by the `Module Record referencingModule`.

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 `referencingModule`, `specifier` does not exist or cannot be created, an exception must be thrown.
- This operation must be idempotent if it completes normally. Each time it is called with a specific `referencingModule`, `specifier` pair as arguments it must return the same `Module Record` instance.

Multiple different `referencingModule`, `specifier` pairs may map to the same `Module Record` instance. The actual mapping semantic is implementation-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.

15.2.1.18 Runtime Semantics: `GetModuleNameSpace` (`module`)

The abstract operation `GetModuleNameSpace` called with argument `module` performs the following steps:

1. Assert: `module` is an instance of a concrete subclass of `Module Record`.
2. Let `namespace` be `module`.\[[\text{Namespace}]\].
3. If `namespace` is `undefined`, then
b. Let `unambiguousNames` be a new empty `List`.
c. For each `name` that is an element of `exportedNames`, do
   i. Let `resolution` be `? module.ResolveExport(name, « »)`.
      ii. If `resolution` is `null`, throw a `SyntaxError` exception.
      iii. If `resolution` is not "ambiguous", append `name` to `unambiguousNames`.
d. Set `namespace` to `ModuleNamespaceCreate(module, unambiguousNames)`.
4. Return `namespace`.

15.2.1.19 Runtime Semantics: `TopLevelModuleEvaluationJob (sourceText, hostDefined)`

A `TopLevelModuleEvaluationJob` with parameters `sourceText` and `hostDefined` is a job that parses, validates, and evaluates `sourceText` as a `Module`.

1. Assert: `sourceText` is an ECMAScript source text (see clause 10).
2. Let `realm` be the current Realm Record.
3. Let `m` be `ParseModule(sourceText, realm, hostDefined)`.
4. If `m` is a `List` of errors, then
   a. Perform `HostReportErrors(m)`.
   b. Return `NormalCompletion(undefined)`.
5. Perform `? m.ModuleDeclarationInstantiation()`.
6. Assert: All dependencies of `m` have been transitively resolved and `m` is ready for evaluation.

NOTE
An implementation may parse a `sourceText` as a `Module`, analyse it for Early Error conditions, and instantiate it prior to the execution of the `TopLevelModuleEvaluationJob` for that `sourceText`. An implementation may also resolve, pre-parse and pre-analyse, and pre-instantiate module dependencies of `sourceText`. However, the reporting of any errors detected by these actions must be deferred until the `TopLevelModuleEvaluationJob` is actually executed.

15.2.1.20 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 $s_l$ be the result of evaluating ModuleItemList.
2. ReturnIfAbrupt($s_l$).
3. Let $s$ be the result of evaluating ModuleItem.
4. Return Completion(UpdateEmpty($s, s_l.[[Value]]$)).

NOTE The value of a ModuleItemList is the value of the last value producing item in the ModuleItemList.

ModuleItem : ImportDeclaration

1. Return NormalCompletion(empty).

15.2.2 Imports

Syntax

ImportDeclaration :
    import ImportClause FromClause ;
    import ModuleSpecifier ;

ImportClause :
    ImportedDefaultBinding
    NamespaceImport
    NamedImports
    ImportedDefaultBinding , NamespaceImport
    ImportedDefaultBinding , NamedImports

ImportedDefaultBinding :
    ImportedBinding

NamespaceImport :
    * as ImportedBinding

NamedImports :
    { }
    { ImportsList }
    { ImportsList , }

FromClause :
    from ModuleSpecifier

ImportsList :
**15.2.2.1 Static Semantics: Early Errors**

*ModuleItem : ImportDeclaration*

- It is a Syntax Error if the BoundNames of *ImportDeclaration* contains any duplicate entries.

**15.2.2.2 Static Semantics: BoundNames**

*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.

15.2.2.3 Static Semantics: ImportEntries

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.

15.2.2.4 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 defauleEntry be the ImportEntry Record {[[ModuleRequest]]: module, [[ImportName]]: "default", [[ModuleName]]: localName}.
3. Return a new List containing defauleEntry.

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 new `List` containing `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 new `List` containing `entry`.

`ImportSpecifier : IdentifierName as ImportedBinding`

1. Let `importName` be the `StringValue` of `IdentifierName`.
2. Let `localName` be the `StringValue` of `ImportedBinding`.
3. Let `entry` be the `ImportEntry Record` `[[ModuleRequest]]: module, [[ImportName]]: importName, `[[LocalName]]: localName`.
4. Return a new `List` containing `entry`.

15.2.2.5 Static Semantics: ModuleRequests

`ImportDeclaration : import ImportClause FromClause ;`

1. Return ModuleRequests of `FromClause`.

`ModuleSpecifier : StringLiteral`

1. Return a `List` containing the `StringValue` of `StringLiteral`.

15.2.3 Exports

Syntax
ExportDeclaration :
   export * FromClause ;  
   export ExportClause FromClause ;  
   export ExportClause ;  
   export VariableStatement[~Yield, ~Await]  
   export Declaration[~Yield, ~Await]  
   export default HoistableDeclaration[~Yield, ~Await, +Default]  
   export default ClassDeclaration[~Yield, ~Await, +Default]  
   export default [lookahead \notin \{ function , async \ [ no LineTerminator here] function ,  
               class ] ] AssignmentExpression[+In, ~Yield, ~Await] ; 

ExportClause :
   { }  
   { ExportsList }  
   { ExportsList , } 

ExportsList :
   ExportSpecifier  
   ExportsList , ExportSpecifier 

ExportSpecifier :
   IdentifierName  
   IdentifierName as IdentifierName 

15.2.3.1 Static Semantics: Early Errors
ExportDeclaration : export ExportClause ;

   • For each IdentifierName n in ReferencedBindings of ExportClause: 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 ExportClause is treated as an IdentifierReference.

15.2.3.2 Static Semantics: BoundNames

ExportDeclaration :
   export * FromClause ;  
   export ExportClause FromClause ;  
   export ExportClause ; 

1. Return a new empty List.
ExportDeclaration : export VariableStatement

1. Return the BoundNames of VariableStatement.

ExportDeclaration : export Declaration

1. Return the BoundNames of Declaration.

ExportDeclaration : export default 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*"».

15.2.3.3 Static Semantics: ExportedBindings

ExportDeclaration :
   export ExportClause FromClause ;
   export * FromClause ;

1. Return a new empty List.

ExportDeclaration : export ExportClause ;

1. Return the ExportedBindings of ExportClause.

ExportDeclaration : export VariableStatement

1. Return the BoundNames of VariableStatement.

ExportDeclaration : export Declaration

1. Return the BoundNames of Declaration.
ExportDeclaration : `export default` HoistableDeclaration
ExportDeclaration : `export default` ClassDeclaration
ExportDeclaration : `export default` AssignmentExpression ;

1. Return the BoundNames of this `ExportDeclaration`.

ExportClause : `{ }`

1. Return a new empty List.

ExportsList : `ExportsList , ExportSpecifier`

1. Let `names` be the ExportedBindings of `ExportsList`.
2. Append to `names` the elements of the ExportedBindings of `ExportSpecifier`.
3. Return `names`.

ExportSpecifier : IdentifierName

1. Return a List containing the StringValue of `IdentifierName`.

ExportSpecifier : `IdentifierName as IdentifierName`

1. Return a List containing the StringValue of the first `IdentifierName`.

15.2.3.4 Static Semantics: ExportedNames

ExportDeclaration : `export * FromClause` ;

1. Return a new empty List.

ExportDeclaration : `export ExportClause FromClause` ;
ExportDeclaration : `export ExportClause` ;

1. Return the ExportedNames of `ExportClause`.

ExportDeclaration : `export VariableStatement`

1. Return the BoundNames of `VariableStatement`.

ExportDeclaration : `export Declaration`

1. Return the BoundNames of `Declaration`.

ExportDeclaration : `export default` HoistableDeclaration
ExportDeclaration : `export default` ClassDeclaration
ExportDeclaration : `export default` AssignmentExpression ;
1. Return « "default" ».

ExportClause : { }

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 containing the StringValue of IdentifierName.

ExportSpecifier : IdentifierName as IdentifierName

1. Return a List containing the StringValue of the second IdentifierName.

15.2.3.5 Static Semantics: ExportEntries

ExportDeclaration : export * FromClause ;

1. Let module be the sole element of ModuleRequests of FromClause.
2. Let entry be the ExportEntry Record {[[ModuleName]]: module, [[ImportName]]: "*", [[LocalName]]: null, [[ExportName]]: null}.
3. Return a new List containing entry.

ExportDeclaration : export ExportClause FromClause ;

1. Let module be the sole element of ModuleRequests of FromClause.
2. Return ExportEntriesForModule of ExportClause with argument module.

ExportDeclaration : export ExportClause ;

1. Return ExportEntriesForModule of ExportClause with argument null.

ExportDeclaration : export VariableStatement

1. Let entries be a new empty List.
2. Let names be the BoundNames of VariableStatement.
3. For each name in names, do
   a. Append the ExportEntry Record {[[ModuleName]]: 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 `name` in `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 new `List` containing 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 new `List` containing the `ExportEntry Record` `{{[ModuleRequest]}: null, [[ImportName]]: null, [[LocalName]]: localName, [[ExportName]]: "default"}`.

ExportDeclaration : `export default AssignmentExpression ;`

1. Let `entry` be the `ExportEntry Record` `{{[ModuleRequest]}: null, [[ImportName]]: null, [[LocalName]]: "*default*", [[ExportName]]: "default"}`.
2. Return a new `List` containing `entry`.

NOTE "*default*" is used within this specification as a synthetic name for anonymous default export values.

15.2.3.6 Static Semantics: `ExportEntriesForModule`

With parameter `module`.

ExportClause : `{ }`

1. Return a new empty `List`.

ExportsList : `ExportsList , ExportSpecifier`

1. Let `specs` be the `ExportEntriesForModule` of `ExportsList` with argument `module`. 
2. Append to \textit{specs} the elements of the ExportEntriesForModule of \textit{ExportSpecifier} with argument \textit{module}.
3. Return \textit{specs}.

\textit{ExportSpecifier} : IdentifierName

1. Let \textit{sourceName} be the StringValue of \textit{IdentifierName}.
2. If \textit{module} is \texttt{null}, then
   a. Let \textit{localName} be \textit{sourceName}.
   b. Let \textit{importName} be \texttt{null}.
3. Else,
   a. Let \textit{localName} be \texttt{null}.
   b. Let \textit{importName} be \textit{sourceName}.
4. Return a new \texttt{List} containing the \textit{ExportEntry Record} \{[[ModuleRequest]]: \textit{module}, [[ImportName]]: \textit{importName}, [[LocalName]]: \textit{localName}, [[ExportName]]: \textit{sourceName} \}.

\textit{ExportSpecifier} : IdentifierName as IdentifierName

1. Let \textit{sourceName} be the StringValue of the first \textit{IdentifierName}.
2. Let \textit{exportName} be the StringValue of the second \textit{IdentifierName}.
3. If \textit{module} is \texttt{null}, then
   a. Let \textit{localName} be \textit{sourceName}.
   b. Let \textit{importName} be \texttt{null}.
4. Else,
   a. Let \textit{localName} be \texttt{null}.
   b. Let \textit{importName} be \textit{sourceName}.
5. Return a new \texttt{List} containing the \textit{ExportEntry Record} \{[[ModuleRequest]]: \textit{module}, [[ImportName]]: \textit{importName}, [[LocalName]]: \textit{localName}, [[ExportName]]: \textit{exportName} \}.

15.2.3.7 Static Semantics: IsConstantDeclaration

\textit{ExportDeclaration} :

\begin{verbatim}
export * FromClause ;
export ExportClause FromClause ;
export ExportClause ;
export default AssignmentExpression ;
\end{verbatim}

1. Return \texttt{false}.

\textbf{NOTE} It is not necessary to treat \texttt{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.
15.2.3.8 Static Semantics: LexicallyScopedDeclarations

ExportDeclaration : export * FromClause ;
ExportDeclaration : export ExportClause FromClause ;
ExportDeclaration : export ExportClause ;
ExportDeclaration : export VariableStatement

1. Return a new empty List.

ExportDeclaration : export Declaration

1. Return a new List containing DeclarationPart of Declaration.

ExportDeclaration : export default HoistableDeclaration

1. Return a new List containing DeclarationPart of HoistableDeclaration.

ExportDeclaration : export default ClassDeclaration

1. Return a new List containing ClassDeclaration.

ExportDeclaration : export default AssignmentExpression ;

1. Return a new List containing this ExportDeclaration.

15.2.3.9 Static Semantics: ModuleRequests

ExportDeclaration : export * FromClause ;
ExportDeclaration : export ExportClause FromClause ;

1. Return the ModuleRequests of FromClause.

ExportDeclaration :

export ExportClause ;
export VariableStatement
export Declaration
export default HoistableDeclaration
export default ClassDeclaration
export default AssignmentExpression ;

1. Return a new empty List.

15.2.3.10 Static Semantics: ReferencedBindings

ExportClause : { }
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` containing the `IdentifierName`.

ExportSpecifier : IdentifierName as IdentifierName

1. Return a `List` containing the first `IdentifierName`.

15.2.3.11 Runtime Semantics: Evaluation

ExportDeclaration :

```
export * FromClause ;
export ExportClause FromClause ;
export ExportClause ;
```

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 the result of BindingClassDeclarationEvaluation of `ClassDeclaration`.
2. ReturnIfAbrupt(`value`).
3. Let `className` be the sole element of BoundNames of `ClassDeclaration`.
4. If `className` is "*default*", then
   a. Let `hasNameProperty` be ? HasOwnProperty(`value`, "name").
   b. If `hasNameProperty` is `false`, perform `SetFunctionName`(`value`, "default").
c. Let \( \textit{env} \) be the \textit{running execution context}'s LexicalEnvironment.

d. Perform \( ? \text{InitializeBoundName}(*\textit{default*}, \textit{value}, \textit{env}) \).

5. Return \texttt{NormalCompletion}(empty).

**ExportDeclaration** : \texttt{export default AssignmentExpression ;}

1. Let \( \textit{rhs} \) be the result of evaluating \texttt{AssignmentExpression}.
2. Let \( \textit{value} \) be \( ? \text{GetValue}(\textit{rhs}) \).
3. If \texttt{IsAnonymousFunctionDefinition(\textit{AssignmentExpression}) is true}, then
   a. Let \( \textit{hasNameProperty} \) be \( ? \text{HasOwnProperty}(\textit{value}, "\textit{name}" \).
   
   b. If \( \textit{hasNameProperty} \) is \textit{false}, perform \texttt{SetFunctionName(\textit{value}, "\textit{default}" \)}

4. Let \( \textit{env} \) be the \textit{running execution context}'s LexicalEnvironment.
5. Perform \( ? \text{InitializeBoundName}(*\textit{default*}, \textit{value}, \textit{env}) \).
6. Return \texttt{NormalCompletion}(empty).

## 16 Error Handling and Language Extensions

An implementation must report most errors at the time the relevant ECMAScript language construct is evaluated. An \textit{early error} is an error that can be detected and reported prior to the evaluation of any construct in the \texttt{Script} containing the error. The presence of an \textit{early error} prevents the evaluation of the construct. An implementation must report early errors in a \texttt{Script} as part of parsing that \texttt{Script} in \texttt{ParseScript}. Early errors in a \texttt{Module} are reported at the point when the \texttt{Module} would be evaluated and the \texttt{Module} is never initialized. Early errors in \texttt{eval} code are reported at the time \texttt{eval} is called and prevent evaluation of the \texttt{eval} code. All errors that are not early errors are runtime errors.

An implementation must report as an \textit{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 16.2, an implementation may extend \texttt{Script} syntax, \texttt{Module} syntax, and regular expression pattern or flag syntax. To permit this, all operations (such as calling \texttt{eval}, using a regular expression literal, or using the \texttt{Function} or \texttt{RegExp} constructor) that are allowed to throw \texttt{SyntaxError} are permitted to exhibit implementation-defined behaviour instead of throwing \texttt{SyntaxError} when they encounter an implementation-defined extension to the script syntax or regular expression pattern or flag syntax.
Except as restricted in 16.2, an implementation may provide additional types, values, objects, properties, and functions beyond those described in this specification. This may cause constructs (such as looking up a variable in the global scope) to have implementation-defined behaviour instead of throwing an error (such as ReferenceError).

An implementation may define behaviour other than throwing RangeError for toFixed, toExponential, and toPrecision when the fractionDigits or precision argument is outside the specified range.

16.1 HostReportErrors (errorList)

HostReportErrors is an implementation-defined abstract operation that allows host environments to report parsing errors, early errors, and runtime errors.

An implementation of HostReportErrors must complete normally in all cases. The default implementation of HostReportErrors is to unconditionally return an empty normal completion.

**NOTE** errorList will be a List of ECMAScript language values. If the errors are parsing errors or early errors, these will always be SyntaxError or ReferenceError objects. Runtime errors, however, can be any ECMAScript value.

16.2 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, 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 non-strict or built-in function objects 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 the following methods must not be extended except as specified in ECMA-402:
  `Object.prototype.toLocaleString`, `Array.prototype.toLocaleString`,
  `Number.prototype.toLocaleString`, `Date.prototype.toLocaleDateString`,
  `Date.prototype.toLocaleTimeString`, `String.prototype.localeCompare`, `%TypedArray%.prototype.toLocaleString`.

- The RegExp pattern grammars in 21.2.1 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` 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.

17 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 lexical environment of the executing program. Others are accessible as initial properties of the `global object` or indirectly as properties of accessible built-in objects.

Unless specified otherwise, a built-in object that is callable as a function is a built-in function object with the characteristics described in 9.3. Unless specified otherwise, the `[[Extensible]]` internal slot of a built-in object initially has the value `true`. Every built-in function object has a `[[Realm]]` internal slot whose value is the `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 9.3.

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` (19.2.3), as the value of its `[[Prototype]]` internal slot.

Unless otherwise specified every built-in prototype object has the Object prototype object, which is the initial value of the expression `Object.prototype` (19.1.3), as the value of its `[[Prototype]]` internal slot, except the Object prototype object itself.

Built-in function objects that are not identified as constructors do not implement the `[[Construct]]` internal method unless otherwise specified in the description of a particular function.

Unless otherwise specified, each built-in function defined in this specification is created as if by calling the `CreateBuiltinFunction` abstract operation (9.3.3).

Every built-in function object, including constructors, has a `length` property whose value is an integer. Unless otherwise specified, this value is equal to the largest number of named arguments shown in the subclause headings for the function description. Optional parameters (which are indicated with brackets: `[ ]`) or rest parameters (which are shown using the form «...name») are not included in the default argument count.

NOTE 2
For example, the function object that is the initial value of the `map` property of the Array prototype object is described under the subclause heading «Array.prototype.map (callbackFn [, thisArg])» which shows the two named arguments callbackFn and thisArg, the latter being optional; therefore the value of the `length` property of that function object is 1.

Unless otherwise specified, the `length` property of a built-in function object has the attributes `{[[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `true` }.

Every built-in function object, including constructors, that is not identified as an anonymous function has a `name` property whose value is a String. Unless otherwise specified, this value is the name that is given to
the function in this specification. For functions that are specified as properties of objects, the name value is the property name string used to access the function. Functions that are specified as get or set accessor functions of built-in properties have "get " or "set " prepended to the property name string. The value of the name property is explicitly specified for each built-in functions whose property key is a Symbol value.

Unless otherwise specified, the name property of a built-in function object, if it exists, has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

Every other data property described in clauses 18 through 26 and in Annex B.2 has the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true } unless otherwise specified.

Every accessor property described in clauses 18 through 26 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.

18 The Global Object

The unique global object is created before control enters any execution context.

The global object does not have a [[Construct]] internal method; it is not possible to use the global object as a constructor with the new operator.

The global object does not have a [[Call]] internal method; it is not possible to invoke the global object as a function.

The value of the [[Prototype]] internal slot of the global object is implementation-dependent.

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the window property of the global object is the global object itself.

18.1 Value Properties of the Global Object

18.1.1 Infinity

The value of Infinity is +∞ (see 6.1.6). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }. 
18.1.2 NaN

The value of NaN is NaN (see 6.1.6). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

18.1.3 undefined

The value of undefined is undefined (see 6.1.1). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

18.2 Function Properties of the Global Object

18.2.1 eval (x)

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. Let calleeRealm be the current Realm Record.
5. Perform ? HostEnsureCanCompileStrings(callerRealm, calleeRealm).

18.2.1.1 Runtime Semantics: PerformEval (x, evalRealm, strictCaller, direct)

The abstract operation PerformEval with arguments x, evalRealm, strictCaller, and direct performs the following steps:

1. Assert: If direct is false, then strictCaller is also false.
2. If Type(x) is not String, return x.
3. Let thisEnvRec be ! GetThisEnvironment().
4. If thisEnvRec is a function Environment Record, then
   a. Let F be thisEnvRec.[[FunctionObject]].
   b. Let inFunction be true.
   c. Let inMethod be thisEnvRec.HasSuperBinding().
   d. If F.[[ConstructorKind]] is "derived", let inDerivedConstructor be true; otherwise, let inDerivedConstructor be false.
5. Else,
   a. Let inFunction be false.
b. Let \( \text{inMethod} \) be \text{false}.

c. Let \( \text{inDerivedConstructor} \) be \text{false}.

6. Let \( \text{script} \) be the ECMAScript code that is the result of parsing \( x \), interpreted as UTF-16 encoded Unicode text as described in 6.1.4, for the goal symbol \( \text{Script} \). If \( \text{inFunction} \) is \text{false}, additional early error rules from 18.2.1.1 are applied. If \( \text{inMethod} \) is \text{false}, additional early error rules from 18.2.1.2 are applied. If \( \text{inDerivedConstructor} \) is \text{false}, additional early error rules from 18.2.1.3 are applied. If the parse fails, throw a \text{SyntaxError} \) or a \text{ReferenceError} exception, depending on the type of the error (but see also clause 16). Parsing and early error detection may be interwoven in an implementation-dependent manner.

7. If \( \text{script} \) Contains \text{ScriptBody} \) is \text{false}, return \text{undefined}.

8. Let \( \text{body} \) be the \text{ScriptBody} of \( \text{script} \).

9. If \( \text{strictCaller} \) is \text{true}, let \( \text{strictEval} \) be \text{true}.

10. Else, let \( \text{strictEval} \) be IsStrict of \( \text{script} \).

11. Let \( \text{ctx} \) be the running execution context.

12. NOTE: If \( \text{direct} \) is \text{true}, \( \text{ctx} \) will be the execution context that performed the \text{direct eval}. If \( \text{direct} \) is \text{false}, \( \text{ctx} \) will be the execution context for the invocation of the \text{eval} \) function.

13. If \( \text{direct} \) is \text{true}, then

   a. Let \( \text{lexEnv} \) be \text{NewDeclarativeEnvironment}(\( \text{ctx} \)'s LexicalEnvironment).

   b. Let \( \text{varEnv} \) be \( \text{ctx} \)'s VariableEnvironment.

14. Else,

   a. Let \( \text{lexEnv} \) be \text{NewDeclarativeEnvironment}(\( \text{evalRealm} \).[[GlobalEnv]]).

   b. Let \( \text{varEnv} \) be \( \text{evalRealm} \).[[GlobalEnv]].

15. If \( \text{strictEval} \) is \text{true}, set \( \text{varEnv} \) to \( \text{lexEnv} \).

16. If \( \text{ctx} \) is not already suspended, suspend \( \text{ctx} \).

17. Let \( \text{evalCxt} \) be a new ECMAScript code execution context.

18. Set the \( \text{evalCxt}'s \) Function to \text{null}.

19. Set the \( \text{evalCxt}'s \) Realm to \( \text{evalRealm} \).

20. Set the \( \text{evalCxt}'s \) ScriptOrModule to \( \text{ctx} \)'s ScriptOrModule.

21. Set the \( \text{evalCxt}'s \) VariableEnvironment to \( \text{varEnv} \).

22. Set the \( \text{evalCxt}'s \) LexicalEnvironment to \( \text{lexEnv} \).

23. Push \( \text{evalCxt} \) on to the execution context stack; \( \text{evalCxt} \) is now the running execution context.

24. Let \( \text{result} \) be \text{EvalDeclarationInstantiation}(\( \text{body} \), \( \text{varEnv} \), \( \text{lexEnv} \), \( \text{strictEval} \)).

25. If \( \text{result}.[[\text{Type}]] \) is normal, then

   a. Set \( \text{result} \) to the result of evaluating \( \text{body} \).

26. If \( \text{result}.[[\text{Type}]] \) is normal and \( \text{result}.[[\text{Value}]] \) is empty, then

   a. Set \( \text{result} \) to \text{NormalCompletion}(\text{undefined}).

27. Suspend \( \text{evalCxt} \) and remove it from the execution context stack.

28. Resume the context that is now on the top of the execution context stack as the running execution context.

29. Return \text{Completion}(\text{result}).
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.

### 18.2.1.1 Additional Early Error Rules for Eval Outside Functions

These static semantics are applied by `PerformEval` when a direct eval call occurs outside of any function.

**ScriptBody** : `StatementList`

- It is a Syntax Error if `StatementList` Contains `NewTarget`.

### 18.2.1.2 Additional Early Error Rules for Eval Outside Methods

These static semantics are applied by `PerformEval` when a direct eval call occurs outside of a `MethodDefinition`.

**ScriptBody** : `StatementList`

- It is a Syntax Error if `StatementList` Contains `SuperProperty`.

### 18.2.1.3 Additional Early Error Rules for Eval Outside Constructor Methods

These static semantics are applied by `PerformEval` when a direct eval call occurs outside of the constructor method of a `ClassDeclaration` or `ClassExpression`.

**ScriptBody** : `StatementList`

- It is a Syntax Error if `StatementList` Contains `SuperCall`.

### 18.2.2 HostEnsureCanCompileStrings( `callerRealm, calleeRealm` )

HostEnsureCanCompileStrings is an implementation-defined abstract operation that 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.

### 18.2.3 Runtime Semantics: EvalDeclarationInstantiation( `body, varEnv, lexEnv, strict` )
When the abstract operation EvalDeclarationInstantiation is called with arguments `body`, `varEnv`, `lexEnv`, and `strict`, the following steps are taken:

1. Let `varNames` be the VarDeclaredNames of `body`.
2. Let `varDeclarations` be the VarScopedDeclarations of `body`.
3. Let `lexEnvRec` be `lexEnv`'s EnvironmentRecord.
4. Let `varEnvRec` be `varEnv`'s EnvironmentRecord.
5. If `strict` is `false`, then
   a. If `varEnvRec` is a global Environment Record, then
      i. For each `name` in `varNames`, do
         1. If `varEnvRec.HasLexicalDeclaration(name)` is `true`, throw a SyntaxError exception.
         2. NOTE: `eval` will not create a global var declaration that would be shadowed by a
global lexical declaration.
   b. Let `thisLex` be `lexEnv`.
   c. Assert: The following loop will terminate.
   d. Repeat, while `thisLex` is not the same as `varEnv`,
      i. Let `thisEnvRec` be `thisLex`'s EnvironmentRecord.
      ii. If `thisEnvRec` 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 `name` in `varNames`, do
            a. If `thisEnvRec.HasBinding(name)` is `true`, then
               i. Throw a SyntaxError exception.
            b. NOTE: Annex B.3.5 defines alternate semantics for the above step.
            iii. Set `thisLex` to `thisLex`'s outer environment reference.
   6. Let `functionsToInitialize` be a new empty List.
   7. Let `declaredFunctionNames` be a new empty List.
   8. For each `d` in `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, or an
            AsyncFunctionDeclaration.
         ii. NOTE: If there are multiple function declarations for the same name, the last declaration
            is used.
         iii. Let `fn` be the sole element of the BoundNames of `d`.
      iv. If `fn` is not an element of `declaredFunctionNames`, then
         1. If `varEnvRec` is a global Environment Record, then
            b. If `fnDefinable` is `false`, throw a TypeError exception.
         2. Append `fn` to `declaredFunctionNames`. 
3. Insert $d$ as the first element of \textit{functionsToInitialize}.

9. NOTE: Annex B.3.3.3 adds additional steps at this point.

10. Let \textit{declaredVarNames} be a new empty \textit{List}.

11. For each $d$ in \textit{varDeclarations}, do
   a. If $d$ is a \textit{VariableDeclaration}, a \textit{ForBinding}, or a \textit{BindingIdentifier}, then
      i. For each String $vn$ in the BoundNames of $d$, do
         1. If $vn$ is not an element of \textit{declaredFunctionNames}, then
            a. If \textit{varEnvRec} is a global \textit{Environment Record}, then
               i. Let $vnDefinable$ be \textit{? varEnvRec}.CanDeclareGlobalVar($vn$).
               ii. If $vnDefinable$ is false, throw a \textbf{TypeError} exception.
            b. If $vn$ is not an element of \textit{declaredVarNames}, then
               i. Append $vn$ to \textit{declaredVarNames}.
   12. NOTE: No abnormal terminations occur after this algorithm step unless \textit{varEnvRec} is a global \textit{Environment Record} and the \textit{global object} is a Proxy exotic object.

13. Let \textit{lexDeclarations} be the LexicallyScopedDeclarations of \textit{body}.

14. For each element $d$ in \textit{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 \textit{true}, then
         1. Perform \textit{? lexEnvRec}.CreateImmutableBinding($dn$, \textit{true}).
      ii. Else,
         1. Perform \textit{? lexEnvRec}.CreateMutableBinding($dn$, \textit{false}).

15. For each \textit{Parse Node} $f$ in \textit{functionsToInitialize}, do
   a. Let $fn$ be the sole element of the BoundNames of $f$.
   b. Let $fo$ be the result of performing InstantiateFunctionObject for $f$ with argument \textit{lexEnv}.
   c. If \textit{varEnvRec} is a global \textit{Environment Record}, then
      i. Perform \textit{? varEnvRec}.CreateGlobalFunctionBinding($fn$, $fo$, \textit{true}).
   d. Else,
      i. Let $bindingExists$ be \textit{varEnvRec}.HasBinding($fn$).
      ii. If $bindingExists$ is \textit{false}, then
         1. Let $status$ be \textit{! varEnvRec}.CreateMutableBinding($fn$, \textit{true}).
         2. Assert: $status$ is not an \textit{abrupt completion} because of validation preceding step 12.
         3. Perform \textit{! varEnvRec}.InitializeBinding($fn$, $fo$).
      iii. Else,
         1. Perform \textit{! varEnvRec}.SetMutableBinding($fn$, $fo$, \textit{false}).

16. For each String $vn$ in \textit{declaredVarNames}, in list order, do
   a. If \textit{varEnvRec} is a global \textit{Environment Record}, then
      i. Perform \textit{? varEnvRec}.CreateGlobalVarBinding($vn$, \textit{true}).
   b. Else,
      i. Let $bindingExists$ be \textit{varEnvRec}.HasBinding($vn$).
ii. If `bindingExists` is `false`, then
   1. Let `status` be `! varEnvRec.CreateMutableBinding(vn, true)`.
   2. Assert: `status` is not an abrupt completion because of validation preceding step 12.
   3. Perform `! varEnvRec.InitializeBinding(vn, undefined)`.

17. Return `NormalCompletion(empty)`.

### NOTE
An alternative version of this algorithm is described in B.3.5.

#### 18.2.2 `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`, `+∞`, or `-∞`, return `false`.
3. Otherwise, return `true`.

#### 18.2.3 `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`.

#### 18.2.4 `parseFloat(string)`

The `parseFloat` function produces a `Number` value dictated by interpretation of the contents of the `string` argument as a decimal literal.

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 a substring of `inputString` consisting of the leftmost code unit that is not a `StrWhiteSpaceChar` and all code units to the right of that code unit. (In other words, remove leading white space.) If `inputString` does not contain any such code units, let `trimmedString` be the empty string.
3. If neither `trimmedString` nor any prefix of `trimmedString` satisfies the syntax of a `StrDecimalLiteral` (see 7.1.3.1), 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 "-", return -0.
   b. Return +0.

7. Return the Number value for `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.

### 18.2.5 `parseInt` ( `string`, `radix` )

The `parseInt` function produces an integer value dictated by interpretation of the contents of the `string` argument according to the specified `radix`. Leading white space in `string` is ignored. If `radix` is `undefined` or 0, it is assumed to be 10 except when the number begins with the 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 a newly created substring of `inputString` consisting of the first code unit that is not a `StrWhiteSpaceChar` and all code units following that code unit. (In other words, remove leading white space.) If `inputString` does not contain any such code unit, let `S` be the empty string.
3. Let `sign` be 1.
4. If `S` is not empty and the first code unit of `S` is 0x002D (HYPHEN-MINUS), let `sign` be -1.
5. If `S` is not empty and the first code unit of `S` is 0x002B (PLUS SIGN) or 0x002D (HYPHEN-MINUS), remove the first code unit from `S`.
6. Let `R` be `? ToInt32(radix)`.
7. Let `stripPrefix` be `true`.
8. If `R ≠ 0`, then
   a. If `R < 2` or `R > 36`, return NaN.
   b. If `R ≠ 16`, let `stripPrefix` be `false`.
9. Else `R = 0`,
   a. Let `R` be 10.
10. If `stripPrefix` is `true`, then
a. If the length of $S$ is at least 2 and the first two code units of $S$ are either "\texttt{0x}" or "\texttt{0X}", remove the first two code units from $S$ and let $R$ be 16.

11. If $S$ contains a code unit that is not a radix-$R$ digit, let $Z$ be the substring of $S$ consisting of all code units before the first such code unit; otherwise, let $Z$ be $S$.

12. If $Z$ is empty, return NaN.

13. Let $\texttt{mathInt}$ be the mathematical integer value that is represented by $Z$ in radix-$R$ notation, using the letters A-Z and a-z for digits with values 10 through 35. (However, if $R$ is 10 and $Z$ contains more than 20 significant digits, every significant digit after the 20th may be replaced by a 0 digit, at the option of the implementation; and if $R$ is not 2, 4, 8, 10, 16, or 32, then $\texttt{mathInt}$ may be an implementation-dependent approximation to the mathematical integer value that is represented by $Z$ in radix-$R$ notation.)

14. If $\texttt{mathInt} = 0$, then
   a. If $\texttt{sign} = -1$, return -0.
   b. Return +0.

15. Let $\texttt{number}$ be the Number value for $\texttt{mathInt}$.

16. Return $\texttt{sign} \times \texttt{number}$.

**NOTE** $\texttt{parseInt}$ may interpret only a leading portion of $\texttt{string}$ as an integer value; it ignores any code units that cannot be interpreted as part of the notation of an integer, and no indication is given that any such code units were ignored.

### 18.2.6 URI Handling Functions

Uniform Resource Identifiers, or URIs, are Strings that identify resources (e.g. web pages or files) and transport protocols by which to access them (e.g. HTTP or FTP) on the Internet. The ECMAScript language itself does not provide any support for using URIs except for functions that encode and decode URIs as described in 18.2.6.2, 18.2.6.3, 18.2.6.4 and 18.2.6.5

**NOTE** Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

### 18.2.6.1 URI Syntax and Semantics

A URI is composed of a sequence of components separated by component separators. The general form is:

$$\textit{Scheme} : \textit{First} / \textit{Second} ; \textit{Third} ? \textit{Fourth}$$

where the italicized names represent components and ";", "/", "?" and "?" are reserved for use as separators. The $\texttt{encodeURI}$ and $\texttt{decodeURI}$ 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 $\texttt{encodeURICComponent}$ and $\texttt{decodeURICComponent}$ 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".

18.2.6.1.1 Runtime Semantics: Encode (string, unescapedSet)

The encoding and escaping process is described by the abstract operation Encode taking two String arguments string and unescapedSet.

1. Let strLen be the number of code units in string.
2. Let R be the empty String.
3. Let k be 0.
4. Repeat,
   a. If k equals strLen, return R.
   b. Let C be the code unit at index k within string.
   c. If C is in unescapedSet, then
      i. Let S be a String containing only the code unit C.
      ii. Set R to a new String value computed by concatenating the previous value of R and S.
   d. Else C is not in unescapedSet,
      i. If the code unit value of C is not less than 0xDC00 and not greater than 0xDFFF, throw a URIError exception.
      ii. If the code unit value of C is less than 0xD800 or greater than 0xDBFF, then
         1. Let V be the code point with the same numeric value as code unit C.
         iii. Else,
            1. Increase k by 1.
            2. If k equals strLen, throw a URIError exception.
            3. Let kChar be the code unit value of the code unit at index k within string.
            4. If kChar is less than 0xDC00 or greater than 0xDFFF, throw a URIError exception.
            5. Let V be UTF16Decode(C, kChar).
      iv. Let Octets be the array of octets resulting by applying the UTF-8 transformation to V, and let L be the array size.
   v. Let j be 0.
   vi. Repeat, while j < L
      1. Let jOctet be the value at index j within Octets.
      2. Let S be a String containing three code units "%XX" where XY are two uppercase hexadecimal digits encoding jOctet.
      3. Set R to a new String value computed by concatenating the previous value of R and S.
      4. Increase j by 1.
   e. Increase k by 1.
18.2.6.1.2 Runtime Semantics: Decode (string, reservedSet)

The unescaping and decoding process is described by the abstract operation Decode taking two String arguments string and reservedSet.

1. Let strLen be the number of code units in string.
2. Let R be the empty String.
3. Let k be 0.
4. Repeat,
   a. If k equals strLen, return R.
   b. Let C be the code unit at index k within string.
   c. If C is not "%", then
      i. Let S be the String containing only the code unit C.
   d. Else C is "%",
      i. Let start be k.
      ii. If k + 2 is greater than or equal to strLen, throw a URIError exception.
      iii. If the code units at 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. Increment k by 2.
   vi. If the most significant bit in B is 0, then
      1. Let C be the code unit with code unit value B.
      2. If C is not in reservedSet, then
         a. Let S be the String containing only the code unit C.
      3. Else C is in reservedSet,
         a. Let S be the substring of string from index start to index k inclusive.
   vii. Else the most significant bit in B is 1,
      1. Let n be the smallest nonnegative integer such that (B << n) & 0x80 is equal to 0.
      2. If n equals 1 or n is greater than 4, throw a URIError exception.
      3. Let Octets be an array of 8-bit integers of size n.
      4. Put B into Octets at index 0.
      5. If k + (3 × (n - 1)) is greater than or equal to strLen, throw a URIError exception.
      7. Repeat, while j < n
         a. Increment k by 1.
         b. If the code unit at index k within string is not "%", throw a URIError exception.
         c. If the code units at index (k + 1) and (k + 2) within string do not represent hexadecimal digits, throw a URIError exception.
d. Let $B$ be the 8-bit value represented by the two hexadecimal digits at index $(k + 1)$ and $(k + 2)$.

e. If the two most significant bits in $B$ are not 10, throw a **URIError** exception.

f. Increment $k$ by 2.

g. Put $B$ into Octets at index $j$.

h. Increment $j$ by 1.

8. Let $V$ be the value obtained by applying the UTF-8 transformation to Octets, that is, from an array of octets into a 21-bit value. If **Octets** does not contain a valid UTF-8 encoding of a Unicode code point, throw a **URIError** exception.

9. If $V < 0x10000$, then

a. Let $C$ be the code unit $V$.

b. If $C$ is not in **reservedSet**, then

   i. Let $S$ be the String containing only the code unit $C$.

c. Else $C$ is in **reservedSet**, then

   i. Let $S$ be the substring of string from index start to index $k$ inclusive.

10. Else $V \geq 0x10000$,

   a. Let $S$ be the String value whose elements are, in order, the elements in **UTF16Encoding**($V$).

e. Set $R$ to a new String value computed by concatenating the previous value of **R** and **S**.

f. Increase $k$ by 1.

---

**NOTE**

This syntax of Uniform Resource Identifiers is based upon RFC 2396 and does not reflect the more recent RFC 3986 which replaces RFC 2396. A formal description and implementation of UTF-8 is given in RFC 3629.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of 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 44**.

**Table 44 (Informative): UTF-8 Encodings**

<table>
<thead>
<tr>
<th>Code Unit Value</th>
<th>Representation</th>
<th>1st Octet</th>
<th>2nd Octet</th>
<th>3rd Octet</th>
<th>4th Octet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 - 0x007F</td>
<td>00000000 0zzzzzz</td>
<td>0zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0080 - 0x07FF</td>
<td>00000000 yyyyzzzz</td>
<td>110yyyyy</td>
<td>10zzzzzz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0xD7FF</td>
<td>xxxxxxxx yyyyzzzz</td>
<td>1110xxxx</td>
<td>10yyyyyy</td>
<td>10zzzzzz</td>
<td></td>
</tr>
<tr>
<td>0xD800 - 0xDBFF</td>
<td>110110vv vvwwwwxx</td>
<td>11110uuu</td>
<td>10uuuuuu</td>
<td>10xxyyyy</td>
<td>10zzzzz</td>
</tr>
</tbody>
</table>
followed by 0xDC00 - 0xDFFF
followed by 110111yy yyzzzzzz
followed by 0xD800 - 0xDBFF
causes URIError
not followed by 0xDC00 - 0xDFFF
followed by 0xDC00 - 0xDFFF
causes URIError
followed by 0xE000 - 0xFFFF
xxxxyyyy yyzzzzzz
1110xxx 10yyyyy 10zzzzzz

Where
\[ uuuuu = vvv + 1 \]
to account for the addition of 0x10000 as in Surrogates, section 3.8, of the Unicode Standard

The range of code unit values 0xD800-0xDBFF is used to encode surrogate pairs; the above transformation combines a UTF-16 surrogate pair into a UTF-32 representation and encode: the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

RFC 3629 prohibits the decoding of invalid UTF-8 octet sequences. For example, the invalid sequence C0 80 must not decode into the code unit 0x0000. Implementations of the Decode algorithm are required to throw a URIError when encountering such invalid sequences.

18.2.6.2 decodeURI (encodedURI)

The decodeURI function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the encodeURI function is replaced with the UTF-16 encoding of the code points that it represents. Escape sequences that could not have been introduced by encodeURI are not replaced.

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.

18.2.6.3 decodeURICOMPONENT (encodedURICOMPONENT)
The `decodeURIComponent` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURIComponent` function is replaced with the UTF-16 encoding of the code points that it represents.

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)`.

### 18.2.6.4 encodeURI ( `uri` )

The `encodeURI` function computes a new version of a UTF-16 encoded (6.1.4) URI in which each instance of certain code points is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the code points.

The `encodeURI` function is the `%encodeURI%` intrinsic object. When the `encodeURI` function is called with one argument `uri`, the following steps are taken:

1. Let `uriString` be `ToString(uri)`.
2. Let `unescapedURISet` be a String containing one instance of each code unit valid in `uriReserved` and `uriUnescaped` plus `"#"`.
3. Return `Encode(uriString, unescapedURISet)`.

**NOTE**
The code unit `"#"` is not encoded to an escape sequence even though it is not a reserved or unescaped URI code point.

### 18.2.6.5 encodeURIComponent ( `uriComponent` )

The `encodeURIComponent` function computes a new version of a UTF-16 encoded (6.1.4) URI in which each instance of certain code points is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the code point.

The `encodeURIComponent` function is the `%encodeURIComponent%` intrinsic object. When the `encodeURIComponent` function is called with one argument `uriComponent`, the following steps are taken:

1. Let `componentString` be `ToString(uriComponent)`.
2. Let `unescapedURIComponentSet` be a String containing one instance of each code unit valid in `uriUnescaped`.
3. Return `Encode(componentString, unescapedURIComponentSet)`. 
18.3 Constructor Properties of the Global Object

18.3.1 Array (...)
See 22.1.1.

18.3.2 ArrayBuffer (...)
See 24.1.2.

18.3.3 Boolean (...)
See 19.3.1.

18.3.4 DataView (...)
See 24.3.2.

18.3.5 Date (...)
See 20.3.2.

18.3.6 Error (...)
See 19.5.1.

18.3.7 EvalError (...)
See 19.5.5.1.

18.3.8 Float32Array (...)
See 22.2.4.

18.3.9 Float64Array (...)
See 22.2.4.
18.3.10 Function (...)  
See 19.2.1.

18.3.11 Int8Array (...)  
See 22.2.4.

18.3.12 Int16Array (...)  
See 22.2.4.

18.3.13 Int32Array (...)  
See 22.2.4.

18.3.14 Map (...)  
See 23.1.1.

18.3.15 Number (...)  
See 20.1.1.

18.3.16 Object (...)  
See 19.1.1.

18.3.17 Proxy (...)  
See 26.2.1.

18.3.18 Promise (...)  
See 25.4.3.

18.3.19 RangeError (...)  
See 19.5.5.2.
18.3.20 ReferenceError (…) 
See 19.5.5.3.

18.3.21 RegExp (…) 
See 21.2.3.

18.3.22 Set (…) 
See 23.2.1.

18.3.23 SharedArrayBuffer (…) 
See 24.2.2.

18.3.24 String (…) 
See 21.1.1.

18.3.25 Symbol (…) 
See 19.4.1.

18.3.26 SyntaxError (…) 
See 19.5.5.4.

18.3.27 TypeError (…) 
See 19.5.5.5.

18.3.28 Uint8Array (…) 
See 22.2.4.

18.3.29 Uint8ClampedArray (…) 
See 22.2.4.
18.3.30 Uint16Array (…)
See 22.2.4.

18.3.31 Uint32Array (…)
See 22.2.4.

18.3.32 URIError (…)
See 19.5.5.6.

18.3.33 WeakMap (…)
See 23.3.1.

18.3.34 WeakSet (…)
See 23.4.

18.4 Other Properties of the Global Object

18.4.1 Atomics
See 24.4.

18.4.2 JSON
See 24.5.

18.4.3 Math
See 20.2.

18.4.4 Reflect
See 26.1.
19 Fundamental Objects

19.1 Object Objects

19.1.1 The Object Constructor

The Object constructor is the %Object% intrinsic object and the initial value of the Object property of the global object. When called as a constructor it creates a new ordinary object. When Object is called as a function rather than as a constructor, it performs a type conversion.

The Object constructor is designed to be subclassable. It may be used as the value of an extends clause of a class definition.

19.1.1.1 Object ([ value ])

When Object function is called with optional argument value, the following steps are taken:

1. If NewTarget is neither undefined nor the active function, then
   a. Return ? OrdinaryCreateFromConstructor(NewTarget, "%ObjectPrototype%")
2. If value is null, undefined or not supplied, return ObjectCreate(%ObjectPrototype%).
3. Return ! ToObject(value).

The length property of the Object constructor function is 1.

19.1.2 Properties of the Object Constructor

The value of the [[Prototype]] internal slot of the Object constructor is the intrinsic object %FunctionPrototype%.

Besides the length property, the Object constructor has the following properties:

19.1.2.1 Object.assign ( target, ...sources )

The assign function is used to copy the values of all of the enumerable own properties from one or more source objects to a target object. When the assign function is called, the following steps are taken:

1. Let to be ? ToObject(target).
2. If only one argument was passed, return to.
3. Let sources be the List of argument values starting with the second argument.
4. For each element nextSource of sources, in ascending index order, do
a. If `nextSource` is `undefined` or `null`, let `keys` be a new empty `List`.

b. Else,
   i. Let `from` be `ToObject(nextSource)`.
   ii. Let `keys` be `from.[[OwnPropertyKeys]]()`.

c. For each element `nextKey` of `keys` in `List` order, do
   i. Let `desc` be `from.[[GetProperty]](nextKey)`.
      ii. If `desc` is not `undefined` and `desc.[[Enumerable]]` is `true`, then
         1. Let `propValue` be `Get(from, nextKey)`.
         2. Perform `Set(to, nextKey, propValue, true)`.

5. Return `to`.

The `length` property of the `assign` function is 2.

19.1.2.2 Object.create ( O, Properties )

The `create` function creates a new object with a specified prototype. When the `create` function is called, the following steps are taken:

1. If `Type(O)` is neither `Object` nor `Null`, throw a `TypeError` exception.
2. Let `obj` be `ObjectCreate(O)`.
3. If `Properties` is not `undefined`, then
4. Return `obj`.

19.1.2.3 Object.defineProperties ( O, Properties )

The `defineProperties` function is used to add own properties and/or update the attributes of existing own properties of an object. When the `defineProperties` function is called, the following steps are taken:


19.1.2.3.1 Runtime Semantics: ObjectDefineProperties ( O, Properties )

The abstract operation `ObjectDefineProperties` with arguments `O` and `Properties` performs the following steps:

1. If `Type(O)` is not `Object`, throw a `TypeError` exception.
2. Let `props` be `ToObject(Properties)`.
3. Let `keys` be `props.[[OwnPropertyKeys]]()`.
4. Let `descriptors` be a new empty `List`.
5. For each element `nextKey` of `keys` in `List` order, 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 `pair` from `descriptors` in list order, do
   a. Let `P` be the first element of `pair`.
   b. Let `desc` be the second element of `pair`.
   c. Perform \( ? \text{DefinePropertyOrThrow}(O, P, desc) \).

7. Return `O`.

### 19.1.2.4 Object.defineProperty ( \( O, P, Attributes \) )

The `defineProperty` function is used to add an own property and/or update the attributes of an existing own property of an object. When the `defineProperty` function is called, the following steps are taken:

1. If `Type(O)` is not `Object`, throw a `TypeError` exception.
2. Let `key` be \( ? \text{ToPropertyKey}(P) \).
3. Let `desc` be \( ? \text{ToPropertyDescriptor}(Attributes) \).
4. Perform \( ? \text{DefinePropertyOrThrow}(O, key, desc) \).
5. Return `O`.

### 19.1.2.5 Object.entries ( \( O \) )

When the `entries` function is called with argument `O`, the following steps are taken:

1. Let `obj` be \( ? \text{ToObject}(O) \).
2. Let `nameList` be \( \text{EnumerableOwnProperties}(obj, "key+value") \).
3. Return `CreateArrayFromList(nameList)`.

### 19.1.2.6 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 \( ? \text{SetIntegrityLevel}(O, "frozen") \).
3. If `status` is `false`, throw a `TypeError` exception.
4. Return `O`.

### 19.1.2.7 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)`.

### 19.1.2.8 `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 `ObjectCreate(%ObjectPrototype%)`.
4. For each element `key` of `ownKeys` in `List` order, do
   a. Let `desc` be `obj.[[GetOwnProperty]](key)`.
   b. Let `descriptor` be `FromPropertyDescriptor(desc)`.
   c. If `descriptor` is not `undefined`, perform `CreateDataProperty(descriptors, key, descriptor)`.
5. Return `descriptors`.

### 19.1.2.9 `Object.getOwnPropertyNames (O)`

When the `getOwnPropertyNames` function is called, the following steps are taken:


### 19.1.2.10 `Object.getOwnPropertySymbols (O)`

When the `getOwnPropertySymbols` function is called with argument `O`, the following steps are taken:


### 19.1.2.10.1 Runtime Semantics: `GetOwnPropertyKeys (O, Type)`

The abstract operation `GetOwnPropertyKeys` is called with arguments `O` and `Type` where `O` is an Object and `Type` is one of the ECMAScript specification types String or Symbol. The following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. Let `keys` be `obj.[[OwnPropertyKeys]]()`.
3. Let `nameList` be a new empty `List`.
4. For each element `nextKey` of `keys` in `List` order, do
   a. If `Type(nextKey)` is `Type`, then
      i. Append `nextKey` as the last element of `nameList`. 
5. Return `CreateArrayFromList(nameList)`.

### 19.1.2.11 Object.getPrototypeOf (O)

When the `getPrototypeOf` function is called with argument `O`, the following steps are taken:

1. Let `obj` be `ToObject(O).
2. Return `obj.[[GetPrototypeOf]]()`.

### 19.1.2.12 Object.is (value1, value2)

When the `is` function is called with arguments `value1` and `value2`, the following steps are taken:

1. Return `SameValue(value1, value2)`.

### 19.1.2.13 Object.isExtensible (O)

When the `isExtensible` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not `Object`, return `false`.
2. Return `IsExtensible(O)`.

### 19.1.2.14 Object.isFrozen (O)

When the `isFrozen` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not `Object`, return `true`.
2. Return `TestIntegrityLevel(O, "frozen")`.

### 19.1.2.15 Object.isSealed (O)

When the `isSealed` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not `Object`, return `true`.
2. Return `TestIntegrityLevel(O, "sealed")`.

### 19.1.2.16 Object.keys (O)

When the `keys` function is called with argument `O`, the following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. Let `nameList` be `EnumerableOwnProperties(obj, "key")`.
3. Return `CreateArrayFromList(nameList)`.
19.1.2.17 Object.preventExtensions (O)

When the `preventExtensions` function is called, the following steps are taken:

1. If `Type(O)` is not `Object`, return `O`.
2. Let `status` be `O.isNaN([[PreventExtensions]]())`.
3. If `status` is `false`, throw a `TypeError` exception.
4. Return `O`.

19.1.2.18 Object.prototype

The initial value of `Object.prototype` is the intrinsic object `%ObjectPrototype%`.

This property has the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

19.1.2.19 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`.

19.1.2.20 Object.setPrototypeOf (O, proto)

When the `setPrototypeOf` function is called with arguments `O` and `proto`, the following steps are taken:

1. Let `O` be `ToObjectCoercible(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.isNaN([[SetPrototypeOf]](proto))`.
5. If `status` is `false`, throw a `TypeError` exception.
6. Return `O`.

19.1.2.21 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)`.
19.1.3 Properties of the Object Prototype Object

The Object prototype object is the intrinsic object `%ObjectPrototype%`. The Object prototype object is an immutable prototype exotic object.

The value of the [[Prototype]] internal slot of the Object prototype object is `null` and the initial value of the [[Extensible]] internal slot is `true`.

19.1.3.1 Object.prototype.constructor

The initial value of `Object.prototype.constructor` is the intrinsic object `%Object%`.

19.1.3.2 Object.prototype.hasOwnProperty (V)

When the `hasOwnProperty` method is called with argument `V`, the following steps are taken:

1. Let `P` be `ToPropertyKey(V)`.
2. Let `O` be `ToObject(this value)`.
3. Return `HasOwnProperty(O, P)`.

**NOTE** The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the `this` value is `undefined` or `null`.

19.1.3.3 Object.prototype.isPrototypeOf (V)

When the `isPrototypeOf` method is called with argument `V`, the following steps are taken:

1. If `Type(V)` is not Object, return `false`.
2. Let `O` be `ToObject(this value)`.
3. Repeat,
   a. Let `V` be `V.[[GetPrototypeOf]]()`.
   b. If `V` is `null`, return `false`.
   c. If `SameValue(O, V)` is `true`, return `true`.

**NOTE** The ordering of steps 1 and 2 preserves the behaviour specified by previous editions of this specification for the case where `V` is not an object and the `this` value is `undefined` or `null`.

19.1.3.4 Object.prototype.propertyIsEnumerable (V)

When the `propertyIsEnumerable` method is called with argument `V`, the following steps are taken:
1. Let \( P \) be ? ToPropertyKey(\( V \)).
2. Let \( O \) be ? ToObject(this value).
3. Let \( desc \) be ? \( O.\)[[GetOwnProperty]](\( P \)).
4. If \( desc \) is undefined, return false.
5. Return \( desc.\)[[Enumerable]].

**NOTE 1** This method does not consider objects in the prototype chain.

**NOTE 2** The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the this value is undefined or null.

### 19.1.3.5 Object.prototype.toLocaleString ([reserved1 [, reserved2 ]])

When the toLocaleString method is called, the following steps are taken:

1. Let \( O \) be the this value.
2. Return ? Invoke(\( O \), "toString").

The optional parameters to this function are not used but are intended to correspond to the parameter pattern used by ECMA-402 toLocaleString functions. Implementations that do not include ECMA-402 support must not use those parameter positions for other purposes.

**NOTE 1** This function provides a generic toLocaleString implementation for objects that have no locale-specific toString behaviour: Array, Number, Date, and Typed Arrays provide their own locale-sensitive toLocaleString methods.

**NOTE 2** ECMA-402 intentionally does not provide an alternative to this default implementation.

### 19.1.3.6 Object.prototype.toString ()

When the toString method is called, the following steps are taken:

1. If the this value is undefined, return "[object Undefined]".
2. If the this value is null, return "[object Null]".
3. Let \( O \) be ! ToObject(this value).
4. Let isArray be ? IsArray(\( O \)).
5. If isArray is true, let builtinTag be "Array".
6. Else if \( O \) is a String exotic object, let builtinTag be "String".
7. Else if \( O \) has a [[ParameterMap]] internal slot, let builtinTag be "Arguments".
8. Else if \( O \) has a [[Call]] internal method, let builtinTag be "Function".
9. Else if \( O \) has an [[ErrorData]] internal slot, let builtinTag be "Error".
10. Else if \( O \) has a [[BooleanData]] internal slot, let builtinTag be "Boolean".
11. Else if $O$ has a $[[\text{NumberData}]]$ internal slot, let $\text{builtinTag}$ be "Number".
12. Else if $O$ has a $[[\text{DateValue}]]$ internal slot, let $\text{builtinTag}$ be "Date".
13. Else if $O$ has a $[[\text{RegExpMatcher}]]$ internal slot, let $\text{builtinTag}$ be "RegExp".
14. Else, let $\text{builtinTag}$ be "Object".
15. Let $tag$ be $\text{Get}(O, @@\text{toStringTag})$.
16. If Type($tag$) is not String, let $tag$ be $\text{builtinTag}$.
17. Return the String that is the result of concatenating "$[\text{object }, tag, \text{ and } "$]."

This function is the %ObjProto_toString% intrinsic object.

NOTE  Historically, this function was occasionally used to access the String value of the $[[\text{Class}]]$ 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 $\text{toString}$ preserves compatibility for legacy code that uses $\text{toString}$ 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 $@@\text{toStringTag}$ in ways that will invalidate the reliability of such legacy type tests.

19.1.3.7  Object.prototype.valueOf ()

When the valueOf method is called, the following steps are taken:

1. Return $\text{ToObject}(this$ value).

This function is the %ObjProto_valueOf% intrinsic object.

19.1.4  Properties of Object Instances

Object instances have no special properties beyond those inherited from the Object prototype object.

19.2  Function Objects

19.2.1  The Function Constructor

The Function constructor is the %Function% intrinsic object and the initial value of the Function property of the global object. When Function is called as a function rather than as a constructor, it creates and initializes a new function object. Thus the function call Function(...) is equivalent to the object creation expression new Function(...) with the same arguments.

The Function 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 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 and AsyncFunction subclasses.

19.2.1.1 Function ($p_1$, $p_2$, ..., $p_n$, $body$)

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 $p_1$, $p_2$, ..., $p_n$, $body$ (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")`

19.2.1.1 Runtime Semantics: CreateDynamicFunction( constructor, newTarget, kind, args )

The abstract operation CreateDynamicFunction is called with arguments constructor, newTarget, kind, and args. constructor is the constructor function that is performing this action, newTarget is the constructor that new was initially applied to, kind is either "normal", "generator", or "async", and args is a List containing the actual argument values that were passed to constructor. 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. Let calleeRealm be the current Realm Record.
5. Perform ? HostEnsureCanCompileStrings(callerRealm, calleeRealm).
6. If newTarget is undefined, set newTarget to constructor.
7. If kind is "normal", then
   a. Let goal be the grammar symbol FunctionBody[~ Yield, ~Await] ·
   b. Let parameterGoal be the grammar symbol FormalParameters[~Yield, ~Await] ·
   c. Let fallbackProto be "%FunctionPrototype%".
8. Else if kind is "generator", then
a. Let \textit{goal} be the grammar symbol \textit{GeneratorBody}.
b. Let \textit{parameterGoal} be the grammar symbol \textit{FormalParameters} \texttt{[+Yield, ~Await]}.
c. Let \textit{fallbackProto} be "\textit{%Generator}%".

9. Else,
   a. Assert: \textit{kind} is "\textit{async}".
   b. Let \textit{goal} be the grammar symbol \textit{AsyncFunctionBody}.
   c. Let \textit{parameterGoal} be the grammar symbol \textit{FormalParameters} \texttt{[~Yield, +Await]}.
   d. Let \textit{fallbackProto} be "\textit{%AsyncFunctionPrototype}%".

10. Let \textit{argCount} be the number of elements in \textit{args}.
11. Let \textit{P} be the empty String.
12. If \textit{argCount} = 0, let \textit{bodyText} be the empty String.
13. Else if \textit{argCount} = 1, let \textit{bodyText} be \textit{args[0]}.
14. Else \textit{argCount} > 1,
   a. Let \textit{firstArg} be \textit{args[0]}.
   b. Set \textit{P} to \texttt{? ToString(firstArg)}.
   c. Let \textit{k} be 1.
   d. Repeat, while \textit{k} < \textit{argCount}-1
      i. Let \textit{nextArg} be \textit{args[k]}.
      ii. Let \textit{nextArgString} be \texttt{? ToString(nextArg)}.
      iii. Set \textit{P} to the result of concatenating the previous value of \textit{P}, the String ",", (a comma), and \textit{nextArgString}.
      iv. Increase \textit{k} by 1.
   e. Let \textit{bodyText} be \textit{args[k]}.

15. Set \textit{bodyText} to \texttt{? ToString(bodyText)}.
16. Let \textit{parameters} be the result of parsing \textit{P}, interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using \textit{parameterGoal} as the \textit{goal symbol}. Throw a \texttt{SyntaxError} exception if the parse fails.
17. Let \textit{body} be the result of parsing \textit{bodyText}, interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using \textit{goal} as the \textit{goal symbol}. Throw a \texttt{SyntaxError} exception if the parse fails.
18. Let \textit{strict} be \texttt{ContainsUseStrict of body}.
19. If any static semantics errors are detected for \textit{parameters} or \textit{body}, throw a \texttt{SyntaxError} or a \texttt{ReferenceError} exception, depending on the type of the error. If \textit{strict} is \texttt{true}, the Early Error rules for \textit{UniqueFormalParameters} : \texttt{FormalParameters} are applied. Parsing and early error detection may be interwoven in an implementation-dependent manner.
20. If \textit{strict} is \texttt{true} and IsSimpleParameterList of \textit{parameters} is \texttt{false}, throw a \texttt{SyntaxError} exception.
21. If any element of the BoundNames of \textit{parameters} also occurs in the LexicallyDeclaredNames of \textit{body}, throw a \texttt{SyntaxError} exception.
22. If \textit{body} Contains \texttt{SuperCall} is \texttt{true}, throw a \texttt{SyntaxError} exception.
23. If \textit{parameters} Contains \texttt{SuperCall} is \texttt{true}, throw a \texttt{SyntaxError} exception.
24. If \textit{body} Contains \texttt{SuperProperty} is \texttt{true}, throw a \texttt{SyntaxError} exception.
25. If \textit{parameters} Contains \texttt{SuperProperty} is \texttt{true}, throw a \texttt{SyntaxError} exception.
26. If `kind` is "generator", then
   a. If `parameters` Contains `YieldExpression` is `true`, throw a `SyntaxError` exception.
27. If `kind` is "async", then
   a. If `parameters` Contains `AwaitExpression` is `true`, throw a `SyntaxError` exception.
28. If `strict` is `true`, then
   a. If `BoundNames` of `parameters` contains any duplicate elements, throw a `SyntaxError` exception.
29. Let `proto` be ? `GetPrototypeFromConstructor`(newTarget, `fallbackProto`).
30. Let `F` be `FunctionAllocate`(proto, strict, kind).
31. Let `realmF` be `F`.[[Realm]].
32. Let `scope` be `realmF`.[[GlobalEnv]].
33. Perform `FunctionInitialize`(F, Normal, parameters, body, `scope`).
34. If `kind` is "generator", then
   a. Let `prototype` be `ObjectCreate`(%GeneratorPrototype%).
      b. Perform `DefinePropertyOrThrow`(F, "prototype", PropertyDescriptor{[[Value]]: prototype, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}).
35. Else if `kind` is "normal", perform `MakeConstructor`(F).
36. NOTE: Async functions are not constructable and do not have a [[Construct]] internal method or a "prototype" property.
37. Perform `SetFunctionName`(F, "anonymous").
38. Return `F`.

NOTE A `prototype` property is created for every non-asyn function created using `CreateDynamicFunction` to provide for the possibility that the function will be used as a constructor.

19.2.2 Properties of the Function Constructor

The Function constructor is itself a built-in function object. The value of the [[Prototype]] internal slot of the Function constructor is the intrinsic object %FunctionPrototype%.

The value of the [[Extensible]] internal slot of the Function constructor is `true`.

The Function constructor has the following properties:

19.2.2.1 Function.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `true` }.

19.2.2.2 Function.prototype
The value of `Function.prototype` is `%FunctionPrototype%`, the intrinsic Function prototype object.

This property has the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

### 19.2.3 Properties of the Function Prototype Object

The Function prototype object is the intrinsic object `%FunctionPrototype%`. The Function prototype object is itself a built-in function object. When invoked, it accepts any arguments and returns `undefined`. It does not have a [[Construct]] internal method so it is not a constructor.

**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.

The value of the [[Prototype]] internal slot of the Function prototype object is the intrinsic object `%ObjectPrototype%`. The initial value of the [[Extensible]] internal slot of the Function prototype object is `true`.

The Function prototype object does not have a `prototype` property.

The value of the `length` property of the Function prototype object is 0.

The value of the `name` property of the Function prototype object is the empty String.

#### 19.2.3.1 `Function.prototype.apply` ( `thisArg`, `argArray` )

When the `apply` method is called on an object `func` with arguments `thisArg` and `argArray`, the following steps are taken:

1. If `IsCallable(func)` is `false`, throw a `TypeError` exception.
2. If `argArray` is `undefined` or `null`, then
   a. Perform `PrepareForTailCall()`.
   b. Return `? Call(func, thisArg)`.
3. Let `argList` be `? CreateListFromArrayLike(argArray)`.
4. Perform `PrepareForTailCall()`.
5. Return `? Call(func, thisArg, argList)`.

**NOTE 1** The `thisArg` value is passed without modification as the `this` value. This is a change from Edition 3, where an `undefined` or `null` `thisArg` is replaced with the `global object` and `ToObject` is applied to all other values and that result is passed as the `this` value. Even though the `thisArg` is passed without modification, non-strict functions still perform these transformations upon entry to the function.

**NOTE 2** If `func` is an arrow function or a `bound function` then the `thisArg` will be ignored by the
function [[Call]] in step 5.

19.2.3.2 `Function.prototype.bind (thisArg, ...args)`

When the `bind` method is called with argument `thisArg` and zero or more `args`, it performs the following steps:

1. Let `Target` be the `this` value.
2. If `IsCallable(Target)` is `false`, throw a `TypeError` exception.
3. Let `args` be a new (possibly empty) `List` consisting of all of the argument values provided after `thisArg` in order.
4. Let `F` be ? `BoundFunctionCreate(Target, thisArg, args)`.
5. Let `targetHasLength` be ? `HasOwnProperty(Target, "length")`.
6. If `targetHasLength` is `true`, then
   a. Let `targetLen` be ? `Get(Target, "length")`.
   b. If `Type(targetLen)` is not `Number`, let `L` be `0`.
   c. Else,
      i. Let `targetLen` be `ToInteger(targetLen)`.
      ii. Let `L` be the larger of `0` and the result of `targetLen` minus the number of elements of `args`.
7. Else, let `L` be `0`.
8. Perform ! `DefinePropertyOrThrow(F, "length", PropertyDescriptor {[[Value]]: L, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true})`.
9. Let `targetName` be ? `Get(Target, "name")`.
10. If `Type(targetName)` is not `String`, let `targetName` be the empty string.
11. Perform `SetFunctionName(F, targetName, "bound")`.
12. Return `F`.

**NOTE 1** Function objects created using `Function.prototype.bind` are exotic objects. They also do not have a `prototype` property.

**NOTE 2** If `Target` is an arrow function or a bound function then the `thisArg` passed to this method will not be used by subsequent calls to `F`.

19.2.3.3 `Function.prototype.call (thisArg, ...args)`

When the `call` method is called on an object `func` with argument, `thisArg` and zero or more `args`, the following steps are taken:

1. If `IsCallable(func)` is `false`, throw a `TypeError` exception.
2. Let `argList` be a new empty `List`.
3. If this method was called with more than one argument, then in left to right order, starting with the second argument, append each argument as the last element of `argList`.
4. Perform `PrepareForTailCall()`.
5. Return `? Call(func, thisArg, argList)`.

**NOTE 1** The `thisArg` value is passed without modification as the `this` value. This is a change from Edition 3, where an `undefined` or `null` `thisArg` is replaced with the `global object` and `ToObject` is applied to all other values and that result is passed as the `this` value. Even though the `thisArg` is passed without modification, non-strict functions still perform these transformations upon entry to the function.

**NOTE 2** If `func` is an arrow function or a bound function then the `thisArg` will be ignored by the function `[[Call]]` in step 5.

### 19.2.3.4 Function.prototype.constructor

The initial value of `Function.prototype.constructor` is the intrinsic object `%Function%`.

### 19.2.3.5 Function.prototype.toString()

When the `toString` method is called on an object `func`, the following steps are taken:

1. If `func` is a Bound Function exotic object, then
   a. Return an implementation-dependent String source code representation of `func`. The representation must conform to the rules below. It is implementation-dependent whether the representation includes bound function information or information about the target function.
2. If `Type(func)` is Object and is either a built-in function object or has an `[[ECMAScriptCode]]` internal slot, then
   a. Return an implementation-dependent String source code representation of `func`. The representation must conform to the rules below.
3. Throw a `TypeError` exception.

**toString** Representation Requirements:

- The string representation must have the syntax of a `FunctionDeclaration, FunctionExpression, GeneratorDeclaration, GeneratorExpression, AsyncFunctionDeclaration, AsyncFunctionExpression, ClassDeclaration, ClassExpression, ArrowFunction, AsyncArrowFunction, or MethodDefinition` depending upon the actual characteristics of the object.
- The use and placement of white space, line terminators, and semicolons within the representation String is implementation-dependent.
- If the object was defined using ECMAScript code and the returned string representation is not in the form of a `MethodDefinition` or `GeneratorMethod` then the representation must be such that if the string is evaluated, using `eval` in a lexical context that is equivalent to the lexical context used to create the original object, it will result in a new functionally equivalent object. In that case the
returned source code must not mention freely any variables that were not mentioned freely by the original function's source code, even if these “extra” names were originally in scope.

- If the implementation cannot produce a source code string that meets these criteria then it must return a string for which `eval` will throw a `SyntaxError` exception.

### 19.2.3.6 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

```javascript
v instanceof F
```

evaluates as

```javascript
F[@@hasInstance](v)
```

A constructor function can control which objects are recognized as its instances by `instanceof` by exposing a different `@@hasInstance` method on the function.

This property is non-writable and non-configurable to prevent tampering that could be used to globally expose the target function of a **bound function**.

### 19.2.4 Function Instances

Every Function instance is an ECMAScript function object and has the internal slots listed in **Table 27**. Function objects created using the `Function.prototype.bind` method (**19.2.3.2**) have the internal slots listed in **Table 28**.

Function instances have the following properties:

#### 19.2.4.1 length

The value of the `length` property is an integer that indicates the typical number of arguments expected by the function. However, the language permits the function to be invoked with some other number of
arguments. The behaviour of a function when invoked on a number of arguments other than the number specified by its **length** property depends on the function. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

### 19.2.4.2 name

The value of the **name** property is a String that is descriptive of the function. The name has no semantic significance but is typically a variable or property name that is used to refer to the function at its point of definition in ECMAScript code. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

Anonymous functions objects that do not have a contextual name associated with them by this specification do not have a **name** own property but inherit the **name** property of `%FunctionPrototype%`.

### 19.2.4.3 prototype

Function instances that can be used as a constructor have a **prototype** property. Whenever such a Function instance is created another ordinary object is also created and is the initial value of the function's **prototype** property. Unless otherwise specified, the value of the **prototype** property is used to initialize the [[Prototype]] internal slot of 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 an `ArrowFunction` do not have a **prototype** property.

### 19.3 Boolean Objects

#### 19.3.1 The Boolean Constructor

The Boolean constructor is the `%Boolean%` intrinsic object and the initial value of the `Boolean` property of the global object. When called as a constructor it creates and initializes a new Boolean object. When `Boolean` is called as a function rather than as a constructor, it performs a type conversion.

The `Boolean` 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.
19.3.1.1 Boolean (value)

When Boolean is called with argument value, the following steps are taken:

1. Let b be ToBoolean(value).
2. If NewTarget is undefined, return b.
3. Let O be ? OrdinaryCreateFromConstructor(NewTarget, "%BooleanPrototype", « [[BooleanData]] »).
4. Set O.[[BooleanData]] to b.
5. Return O.

19.3.2 Properties of the Boolean Constructor

The value of the [[Prototype]] internal slot of the Boolean constructor is the intrinsic object %FunctionPrototype%.

The Boolean constructor has the following properties:

19.3.2.1 Boolean.prototype

The initial value of Boolean.prototype is the intrinsic object %BooleanPrototype%.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.3.3 Properties of the Boolean Prototype Object

The Boolean prototype object is the intrinsic object %BooleanPrototype%. The Boolean prototype object is an ordinary object. The Boolean prototype is itself a Boolean object; it has a [[BooleanData]] internal slot with the value false.

The value of the [[Prototype]] internal slot of the Boolean prototype object is the intrinsic object %ObjectPrototype%.

The abstract operation thisBooleanValue(value) performs the following steps:

1. If Type(value) is Boolean, return value.
2. If Type(value) is Object and value has a [[BooleanData]] internal slot, then
   a. Assert: value.[[BooleanData]] is a Boolean value.
   b. Return value.[[BooleanData]].
3. Throw a TypeError exception.

19.3.3.1 Boolean.prototype.constructor
The initial value of `Boolean.prototype.constructor` is the intrinsic object `%Boolean%.

19.3.3.2 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".

19.3.3.3 Boolean.prototype.valueOf ()

The following steps are taken:

1. Return `? thisBooleanValue(this value)`.

19.3.4 Properties of Boolean Instances

Boolean instances are ordinary objects that inherit properties from the Boolean prototype object. Boolean instances have a `[[BooleanData]]` internal slot. The `[[BooleanData]]` internal slot is the Boolean value represented by this Boolean object.

19.4 Symbol Objects

19.4.1 The Symbol Constructor

The Symbol constructor is the `%Symbol%` intrinsic object and the initial value of the `Symbol` property of the global object. When `Symbol` is called as a function, it returns a new Symbol value.

The `Symbol` constructor is not intended to be used with the `new` operator or to be subclassed. It may be used as the value of an `extends` clause of a class definition but a `super` call to the `Symbol` constructor will cause an exception.

19.4.1.1 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`. 
19.4.2 Properties of the Symbol Constructor

The value of the [[Prototype]] internal slot of the Symbol constructor is the intrinsic object %FunctionPrototype%.

The Symbol constructor has the following properties:

19.4.2.1 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 45.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Key]]</code></td>
<td>A String</td>
<td>A string key used to globally identify a Symbol.</td>
</tr>
<tr>
<td><code>[[Symbol]]</code></td>
<td>A Symbol</td>
<td>A symbol that can be retrieved from any realm.</td>
</tr>
</tbody>
</table>

19.4.2.2 Symbol.hasInstance

The initial value of `Symbol.hasInstance` is the well-known symbol `@@hasInstance` (Table 1).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.4.2.3 Symbol.isConcatSpreadable

The initial value of `Symbol.isConcatSpreadable` is the well-known symbol `@@isConcatSpreadable` (Table 1).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.
19.4.2.4 Symbol.iterator

The initial value of `Symbol.iterator` is the well-known symbol `@@iterator` (Table 1).

This property has the attributes `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false`.

19.4.2.5 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 19.4.2.1), 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`.

19.4.2.6 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`.

19.4.2.7 Symbol.prototype

The initial value of `Symbol.prototype` is the intrinsic object `%SymbolPrototype%`.

This property has the attributes `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false`.

19.4.2.8 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`.

19.4.2.9 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`.

19.4.2.10 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 }.

### 19.4.2.11 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 }.

### 19.4.2.12 Symbol.toPrimitive

The initial value of Symbol.toPrimitive is the well-known symbol @@toPrimitive (Table 1).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 19.4.2.13 Symbol.toStringTag

The initial value of Symbol.toStringTag is the well-known symbol @@toStringTag (Table 1).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 19.4.2.14 Symbol.unscopables

The initial value of Symbol.unscopables is the well-known symbol @@unscopables (Table 1).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 19.4.3 Properties of the Symbol Prototype Object

The Symbol prototype object is the intrinsic object %SymbolPrototype%. The Symbol prototype object is an ordinary object. It is not a Symbol instance and does not have a [[SymbolData]] internal slot.

The value of the [[Prototype]] internal slot of the Symbol prototype object is the intrinsic object %ObjectPrototype%.

#### 19.4.3.1 Symbol.prototype.constructor

The initial value of Symbol.prototype.constructor is the intrinsic object %Symbol%.

#### 19.4.3.2 Symbol.prototype.toString ()

The following steps are taken:

1. Let s be the this value.
2. If Type(s) is Symbol, let sym be s.
3. Else,
   a. If Type(s) is not Object, throw a TypeError exception.
   b. If s does not have a [[SymbolData]] internal slot, throw a TypeError exception.
   c. Let sym be s.[[SymbolData]].
4. Return SymbolDescriptiveString(sym).

19.4.3.2.1 Runtime Semantics: SymbolDescriptiveString ( sym )

When the abstract operation SymbolDescriptiveString is called with argument sym, the following steps are taken:

1. Assert: Type(sym) is Symbol.
2. Let desc be sym's [[Description]] value.
3. If desc is undefined, let desc be the empty string.
4. Assert: Type(desc) is String.
5. Return the result of concatenating the strings "Symbol(" desc, and ")".

19.4.3.3 Symbol.prototype.valueOf ()

The following steps are taken:

1. Let s be the this value.
2. If Type(s) is Symbol, return s.
3. If Type(s) is not Object, throw a TypeError exception.
4. If s does not have a [[SymbolData]] internal slot, throw a TypeError exception.
5. Return s.[[SymbolData]].

19.4.3.4 Symbol.prototype [ @@toPrimitive ] ( hint )

This function is called by ECMAScript language operators to convert a Symbol object to a primitive value. The allowed values for hint are "default", "number", and "string".

When the @@toPrimitive method is called with argument hint, the following steps are taken:

1. Let s be the this value.
2. If Type(s) is Symbol, return s.
3. If Type(s) is not Object, throw a TypeError exception.
4. If s does not have a [[SymbolData]] internal slot, throw a TypeError exception.
5. Return s.[[SymbolData]].

The value of the name property of this function is "[Symbol.toPrimitive]".
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

19.4.3.5 Symbol.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "Symbol".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

19.4.4 Properties of Symbol Instances

Symbol instances are ordinary objects that inherit properties from the Symbol prototype object. Symbol instances have a [[SymbolData]] internal slot. The [[SymbolData]] internal slot is the Symbol value represented by this Symbol object.

19.5 Error Objects

Instances of Error objects are thrown as exceptions when runtime errors occur. The Error objects may also serve as base objects for user-defined exception classes.

19.5.1 The Error Constructor

The Error constructor is the %Error% intrinsic object and the initial value of the Error property of the global object. When Error is called as a function rather than as a constructor, it creates and initializes a new Error object. Thus the function call Error(...) is equivalent to the object creation expression new Error(...) with the same arguments.

The Error 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 Error behaviour must include a super call to the Error constructor to create and initialize subclass instances with an [[ErrorData]] internal slot.

19.5.1.1 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, "%ErrorPrototype", « [[ErrorData]] »).
3. If message is not undefined, then
   a. Let msg be ? ToString(message).
b. Let \( \text{msgDesc} \) be the PropertyDescriptor\([[[\text{Value}}]: \text{msg}, [[[\text{Writable}}]: \text{true}, [[[\text{Enumerable}}]: \text{false}, [[[\text{Configurable}}]: \text{true}]].

c. Perform !\text{DefinePropertyOrThrow}(O, "\text{message}", \text{msgDesc}).

4. Return \( O \).

### 19.5.2 Properties of the Error Constructor

The value of the [[Prototype]] internal slot of the Error constructor is the intrinsic object %FunctionPrototype%.

The Error constructor has the following properties:

#### 19.5.2.1 Error.prototype

The initial value of \( \text{Error.prototype} \) is the intrinsic object %ErrorPrototype%.

This property has the attributes \{ [[Writable]]: \text{false}, [[Enumerable]]: \text{false}, [[Configurable]]: \text{false} \}.

### 19.5.3 Properties of the Error Prototype Object

The Error prototype object is the intrinsic object %ErrorPrototype%. The Error prototype object is an ordinary object. It is not an Error instance and does not have an [[ErrorData]] internal slot.

The value of the [[Prototype]] internal slot of the Error prototype object is the intrinsic object %ObjectPrototype%.

#### 19.5.3.1 Error.prototype.constructor

The initial value of \( \text{Error.prototype.constructor} \) is the intrinsic object %Error%.

#### 19.5.3.2 Error.prototype.message

The initial value of \( \text{Error.prototype.message} \) is the empty String.

#### 19.5.3.3 Error.prototype.name

The initial value of \( \text{Error.prototype.name} \) is "Error".

#### 19.5.3.4 Error.prototype.toString ()

The following steps are taken:
1. Let \( O \) be the `this` value.
2. If `Type(O)` is not `Object`, throw a `TypeError` exception.
3. Let `name` be `? Get(O, "name")`.
4. If `name` is `undefined`, let `name` be "Error"; otherwise let `name` be `? ToString(name)`.
5. Let `msg` be `? Get(O, "message")`.
6. If `msg` is `undefined`, let `msg` be the empty String; otherwise let `msg` be `? ToString(msg)`.
7. If `name` is the empty String, return `msg`.
8. If `msg` is the empty String, return `name`.
9. Return the result of concatenating `name`, the code unit 0x003A (COLON), the code unit 0x0020 (SPACE), and `msg`.

### 19.5.4 Properties of Error Instances

Error instances are ordinary objects that inherit properties from the Error prototype object and have an `[[ErrorData]]` internal slot whose value is `undefined`. The only specified uses of `[[ErrorData]]` is to identify Error and `NativeError` instances as Error objects within `Object.prototype.toString`.

### 19.5.5 Native Error Types Used in This Standard

A new instance of one of the `NativeError` objects below is thrown when a runtime error is detected. All of these objects share the same structure, as described in 19.5.6.

#### 19.5.5.1 EvalError

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

#### 19.5.5.2 RangeError

Indicates a value that is not in the set or range of allowable values.

#### 19.5.5.3 ReferenceError

Indicate that an invalid reference value has been detected.

#### 19.5.5.4 SyntaxError

Indicates that a parsing error has occurred.

#### 19.5.5.5 TypeError
TypeError is used to indicate an unsuccessful operation when none of the other `NativeError` objects are an appropriate indication of the failure cause.

### 19.5.6 URIError

Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition.

### 19.5.6 `NativeError` Object Structure

When an ECMAScript implementation detects a runtime error, it throws a new instance of one of the `NativeError` objects defined in 19.5.5. Each of these objects has the structure described below, differing only in the name used as the constructor name instead of `NativeError`, in the `name` property of the prototype object, and in the implementation-defined `message` property of the prototype object.

For each error object, references to `NativeError` in the definition should be replaced with the appropriate error object name from 19.5.5.

### 19.5.6.1 `NativeError` Constructors

When a `NativeError` constructor is called as a function rather than as a constructor, it creates and initializes a new `NativeError` object. 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.

Each `NativeError` 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 `NativeError` behaviour must include a `super` call to the `NativeError` constructor to create and initialize subclass instances with an `[[ErrorData]]` internal slot.

#### 19.5.6.1.1 `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, "%NativeErrorPrototype%", « [[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 "%EvalErrorPrototype%", "%RangeErrorPrototype%", "%ReferenceErrorPrototype%", "%SyntaxErrorPrototype%", "%TypeErrorPrototype%", or "%URIErrorPrototype%" corresponding to which `NativeError` constructor is being defined.

19.5.6.2 Properties of the `NativeError` Constructors

The value of the [[Prototype]] internal slot of a `NativeError` constructor is the intrinsic object `%Error%.

Each `NativeError` constructor has a `name` property whose value is the String value `"NativeError"`.

Each `NativeError` constructor has the following properties:

19.5.6.2.1 `NativeError.prototype`

The initial value of `NativeError.prototype` is a `NativeError` prototype object (19.5.6.3). Each `NativeError` constructor has a distinct prototype object.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.5.6.3 Properties of the `NativeError` Prototype Objects

Each `NativeError` prototype object is an ordinary object. It is not an Error instance and does not have an [[ErrorData]] internal slot.

The value of the [[Prototype]] internal slot of each `NativeError` prototype object is the intrinsic object `%ErrorPrototype%`.

19.5.6.3.1 `NativeError.prototype.constructor`

The initial value of the `constructor` property of the prototype for a given `NativeError` constructor is the corresponding intrinsic object `%NativeError%` (19.5.6.1).

19.5.6.3.2 `NativeError.prototype.message`

The initial value of the `message` property of the prototype for a given `NativeError` constructor is the empty String.

19.5.6.3.3 `NativeError.prototype.name`
The initial value of the `name` property of the prototype for a given `NativeError` constructor is a String consisting of the name of the constructor (the name used instead of `NativeError`).

19.5.6.4 Properties of `NativeError` Instances

`NativeError` instances are ordinary objects that inherit properties from their `NativeError` prototype object and have an `[[ErrorData]]` internal slot whose value is `undefined`. The only specified use of `[[ErrorData]]` is by `Object.prototype.toString(19.1.3.6)` to identify Error or `NativeError` instances.

20 Numbers and Dates

20.1 Number Objects

20.1.1 The Number Constructor

The Number constructor is the `%Number%` intrinsic object and the initial value of the `Number` property of the global object. When called as a constructor, it creates and initializes a new Number object. When `Number` is called as a function rather than as a constructor, it performs a type conversion.

The `Number` 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.

20.1.1.1 Number (value)

When `Number` is called with argument `value`, the following steps are taken:

1. If no arguments were passed to this function invocation, let `n` be `+0`.
2. Else, let `n` be `? ToNumber(value)`.
3. If NewTarget is `undefined`, return `n`.
4. Let `O` be `? OrdinaryCreateFromConstructor(NewTarget, "%NumberPrototype", « [[NumberData]] »)`.
5. Set `O.[[NumberData]]` to `n`.
6. Return `O`.

20.1.2 Properties of the Number Constructor
The value of the [[Prototype]] internal slot of the Number constructor is the intrinsic object %FunctionPrototype%.

The Number constructor has the following properties:

### 20.1.2.1 Number.EPSILON

The value of Number.EPSILON is the difference between 1 and the smallest value greater than 1 that is representable as a Number value, which is approximately $2.2204460492503130808472633361816 \times 10^{-16}$.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 20.1.2.2 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.

### 20.1.2.3 Number.isInteger (number)

When Number.isInteger is called with one argument number, the following steps are taken:

1. If Type(number) is not Number, return false.
2. If number is NaN, $+\infty$, or $-\infty$, return false.
3. Let integer be ToInteger(number).
4. If integer is not equal to number, return false.
5. Otherwise, return true.

### 20.1.2.4 Number.isNaN (number)

When Number.isNaN is called with one argument number, the following steps are taken:

1. If Type(number) is not Number, return false.
2. If number is NaN, return true.
3. Otherwise, return false.

**NOTE** This function differs from the global isNaN function (18.2.3) in that it does not convert its argument to a Number before determining whether it is NaN.
20.1.2.5 Number.isSafeInteger (number)

When `Number.isSafeInteger` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not Number, return `false`.
2. If `number` is `NaN`, `+∞`, or `-∞`, return `false`.
3. Let `integer` be `ToInteger(number)`.
4. If `integer` is not equal to `number`, return `false`.
5. If `abs(integer) ≤ 2^{53} - 1`, return `true`.
6. Otherwise, return `false`.

20.1.2.6 Number.MAX_SAFE_INTEGER

**NOTE** The value of `Number.MAX_SAFE_INTEGER` is the largest integer `n` such that `n` and `n + 1` are both exactly representable as a Number value.

The value of `Number.MAX_SAFE_INTEGER` is `9007199254740991` (2^{53} - 1).

This property has the attributes `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.

20.1.2.7 Number.MAX_VALUE

The value of `Number.MAX_VALUE` is the largest positive finite value of the Number type, which is approximately `1.7976931348623157 × 10^{308}`.

This property has the attributes `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.

20.1.2.8 Number.MIN_SAFE_INTEGER

**NOTE** The value of `Number.MIN_SAFE_INTEGER` is the smallest integer `n` such that `n` and `n - 1` are both exactly representable as a Number value.

The value of `Number.MIN_SAFE_INTEGER` is `-9007199254740991` (-2^{53} - 1).

This property has the attributes `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.

20.1.2.9 Number.MIN_VALUE

The value of `Number.MIN_VALUE` is the smallest positive value of the Number type, which is approximately `5 × 10^{-324}`.

In the IEEE 754-2008 double precision binary representation, the smallest possible value is a denormalized number. If an implementation does not support denormalized values, the value of...
**Number.MIN_VALUE** must be the smallest non-zero positive value that can actually be represented by the implementation.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 20.1.2.10 Number.NaN

The value of **Number.NaN** is **NaN**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 20.1.2.11 Number.NEGATIVE_INFINITY

The value of **Number.NEGATIVE_INFINITY** is **-∞**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 20.1.2.12 Number.parseFloat (**string**) 

The value of the **Number.parseFloat** data property is the same built-in function object that is the value of the **parseFloat** property of the global object defined in 18.2.4.

### 20.1.2.13 Number.parseInt (**string, radix**) 

The value of the **Number.parseInt** data property is the same built-in function object that is the value of the **parseInt** property of the global object defined in 18.2.5.

### 20.1.2.14 Number.POSITIVE_INFINITY

The value of **Number.POSITIVE_INFINITY** is **+∞**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 20.1.2.15 Number.prototype

The initial value of **Number.prototype** is the intrinsic object %NumberPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 20.1.3 Properties of the Number Prototype Object

The Number prototype object is the intrinsic object %NumberPrototype%. The Number prototype object is an ordinary object. The Number prototype is itself a Number object; it has a [[NumberData]] internal slot.
with the value +0.

The value of the [[Prototype]] internal slot of the Number prototype object is the intrinsic object %ObjectPrototype%.

Unless explicitly stated otherwise, the methods of the Number prototype object defined below are not generic and the this value passed to them must be either a Number value or an object that has a [[NumberData]] internal slot that has been initialized to a Number value.

The abstract operation thisNumberValue(value) performs the following steps:

1. If Type(value) is Number, return value.
2. If Type(value) is Object and value has a [[NumberData]] internal slot, then
   a. Assert: value.[[NumberData]] is a Number value.
   b. Return value.[[NumberData]].
3. Throw a TypeError exception.

The phrase “this Number value” within the specification of a method refers to the result returned by calling the abstract operation thisNumberValue with the this value of the method invocation passed as the argument.

20.1.3.1 Number.prototype.constructor

The initial value of Number.prototype.constructor is the intrinsic object %Number%.

20.1.3.2 Number.prototype.toExponential (fractionDigits)

Return a String containing this Number value represented in decimal exponential notation with one digit before the significand's decimal point and fractionDigits digits after the significand's decimal point. If fractionDigits is undefined, include as many significand digits as necessary to uniquely specify the Number (just like in toString except that in this case the Number is always output in exponential notation).

Specifically, perform the following steps:

1. Let x be ? thisNumberValue(this value).
2. Let f be ? ToInteger(fractionDigits).
3. Assert: f is 0, when fractionDigits is undefined.
4. If x is NaN, return the String "NaN".
5. Let s be the empty String.
6. If x < 0, then
   a. Let s be "-".
   b. Let x be -x.
7. If x = +\infty, then
   a. Return the concatenation of the Strings s and "Infinity".
8. If \( f < 0 \) or \( f > 20 \), throw a `RangeError` exception. However, an implementation is permitted to extend the behaviour of `toExponential` for values of \( f \) less than 0 or greater than 20. In this case `toExponential` would not necessarily throw `RangeError` for such values.

9. If \( x = 0 \), then
   a. Let \( m \) be the String consisting of \( f+1 \) occurrences of the code unit 0x0030 (DIGIT ZERO).
   b. Let \( e \) be 0.

10. Else \( x \neq 0 \),
    a. If `fractionDigits` is not `undefined`, then
       i. Let \( e \) and \( n \) be integers such that \( 10^f \leq n < 10^{f+1} \) and for which the exact mathematical value of \( n \times 10^{-f} - x \) is as close to zero as possible. If there are two such sets of \( e \) and \( n \), pick the \( e \) and \( n \) for which \( n \times 10^{-f} \) is larger.
    b. Else `fractionDigits` is `undefined`,
       i. Let \( e \), \( n \), and \( f \) be integers such that \( f \geq 0 \), \( 10^f \leq n < 10^{f+1} \), the Number value for \( n \times 10^{-f} \) is \( x \), and \( f \) is as small as possible. Note that the decimal representation of \( n \) has \( f+1 \) digits, \( n \) is not divisible by 10, and the least significant digit of \( n \) is not necessarily uniquely determined by these criteria.
    c. Let \( m \) be the String consisting of the digits of the decimal representation of \( n \) (in order, with no leading zeroes).

11. If \( f \neq 0 \), then
    a. Let \( a \) be the first element of \( m \), and let \( b \) be the remaining \( f \) elements of \( m \).
    b. Let \( m \) be the concatenation of the three Strings \( a \), " . ", and \( b \).

12. If \( e = 0 \), then
    a. Let \( c \) be " + ".
    b. Let \( d \) be " 0 ".

13. Else,
    a. If \( e > 0 \), let \( c \) be " + ".
    b. Else \( e \leq 0 \),
       i. Let \( c \) be " - ".
       ii. Let \( e \) be -\( e \).
    c. Let \( d \) be the String consisting of the digits of the decimal representation of \( e \) (in order, with no leading zeroes).

14. Let \( m \) be the concatenation of the four Strings \( m \), " e ", \( c \), and \( d \).

15. Return the concatenation of the Strings \( s \) and \( m \).

If the `toExponential` method is called with more than one argument, then the behaviour is implementation-defined (see clause 17).

**NOTE**

For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 10.b.i be used as a guideline:
1. Let $e$, $n$, and $f$ be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, the Number value for $n \times 10^e$ is $x$, and $f$ is as small as possible. If there are multiple possibilities for $n$, choose the value of $n$ for which $n \times 10^e$ is closest in value to $x$. If there are two such possible values of $n$, choose the one that is even.

### 20.1.3.3 Number.prototype.toFixed (fractionDigits)

**NOTE 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 `ToInteger`(`fractionDigits`). (If `fractionDigits` is `undefined`, this step produces the value 0.)
3. If $f < 0$ or $f > 20$, throw a `RangeError` exception. However, an implementation is permitted to extend the behaviour of `toFixed` for values of $f$ less than 0 or greater than 20. In this case `toFixed` would not necessarily throw `RangeError` for such values.
4. If $x$ is `NaN`, return the String "NaN".
5. Let $s$ be the empty String.
6. If $x < 0$, then
   a. Let $s$ be "-".
   b. Let $x$ be -$x$.
7. If $x \geq 10^{21}$, then
   a. Let $m$ be `toString($x$).
8. Else $x < 10^{21}$,
   a. Let $n$ be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such $n$, pick the larger $n$.
   b. If $n = 0$, let $m$ be the String "0". Otherwise, let $m$ be the String consisting of the digits of the decimal representation of $n$ (in order, with no leading zeroes).
   c. If $f \neq 0$, then
      i. Let $k$ be the number of elements in $m$.
      ii. If $k \leq f$, then
         1. Let $z$ be the String consisting of $f+1-k$ occurrences of the code unit 0x0030 (DIGIT ZERO).
         2. Let $m$ be the concatenation of Strings $z$ and $m$.
         3. Let $k$ be $f + 1$.
         iii. Let $a$ be the first $k-f$ elements of $m$, and let $b$ be the remaining $f$ elements of $m$.
      iv. Let $m$ be the concatenation of the three Strings $a$, ", ", and $b$.
9. Return the concatenation of the Strings $s$ and $m$. 

If the `toFixed` method is called with more than one argument, then the behaviour is implementation-defined (see clause 17).

NOTE 2
The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent number values. For example,

\[(1000000000000000128).toString()\] returns "1000000000000000100", while 
\[(1000000000000000128).toFixed(0)\] returns "1000000000000000128".

## 20.1.3.4 Number.prototype.toLocaleString ( [ reserved1 [ , reserved2 ] ] )

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the `Number.prototype.toLocaleString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleString` method is used.

Produces a String value that represents this Number value formatted according to the conventions of the host environment’s current locale. This function is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as `toString`.

The 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.

## 20.1.3.5 Number.prototype.toPrecision ( `precision` )

Return a String containing this Number value represented either in decimal exponential notation with one digit before the significand’s decimal point and `precision`-1 digits after the significand’s decimal point or in decimal fixed notation with `precision` significant digits. If `precision` is `undefined`, call `ToString` 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 `? ToInteger(precision)`.
4. If \( x \) is `NaN`, return the String "`NaN`".
5. Let \( s \) be the empty String.
6. If \( x < 0 \), then
   a. Let \( s \) be code unit `0x002D` (HYPHEN-MINUS).
   b. Let \( x \) be `-x`.
7. If \( x = +\infty \), then
   a. Return the String that is the concatenation of \( s \) and "`Infinity`".
8. If $p < 1$ or $p > 21$, throw a `RangeError` exception. However, an implementation is permitted to extend the behaviour of `toPrecision` for values of $p$ less than 1 or greater than 21. In this case `toPrecision` would not necessarily throw `RangeError` for such values.

9. If $x = 0$, then
   a. Let $m$ be the String consisting of $p$ occurrences of the code unit 0x0030 (DIGIT ZERO).
   b. Let $e$ be 0.

10. Else $x \neq 0$,
    a. Let $e$ and $n$ be integers such that $10^{p-1} \leq n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} \times x$ is as close to zero as possible. If there are two such sets of $e$ and $n$, pick the $e$ and $n$ for which $n \times 10^{e-p+1}$ is larger.
    b. Let $m$ be the String consisting of the digits of the decimal representation of $n$ (in order, with no leading zeroes).
    c. If $e < -6$ or $e \geq p$, then
       i. Assert: $e \neq 0$.
       ii. If $p \neq 1$, then
           1. Let $a$ be the first element of $m$, and let $b$ be the remaining $p-1$ elements of $m$.
           2. Let $m$ be the concatenation of $a$, ".", and $b$.
       iii. If $e > 0$, then
            1. Let $c$ be code unit 0x002B (PLUS SIGN).
       iv. Else $e < 0$,
            1. Let $c$ be code unit 0x002D (HYPHEN-MINUS).
            2. Let $e$ be -$e$.
       v. Let $d$ be the String consisting of the digits of the decimal representation of $e$ (in order, with no leading zeroes).
       vi. Return the concatenation of $s$, $m$, code unit 0x0065 (LATIN SMALL LETTER E), $c$, and $d$.

11. If $e = p-1$, return the concatenation of the Strings $s$ and $m$.

12. If $e \geq 0$, then
    a. Let $m$ be the concatenation of the first $e+1$ elements of $m$, the code unit 0x002E (FULL STOP), and the remaining $p- (e+1)$ elements of $m$.

13. Else $e < 0$,
    a. Let $m$ be the String formed by the concatenation of code unit 0x0030 (DIGIT ZERO), code unit 0x002E (FULL STOP), -$e$ occurrences of code unit 0x0030 (DIGIT ZERO), and the String $m$.

14. Return the String that is the concatenation of $s$ and $m$.

If the `toPrecision` method is called with more than one argument, then the behaviour is implementation-defined (see clause 17).

### 20.1.3.6 Number.prototype.toString ([ `radix` ])

**NOTE** The optional `radix` should be an integer value in the inclusive range 2 to 36. If `radix` not
The following steps are performed:

1. Let $x$ be $\text{thisNumberValue}(\text{this} \text{ value})$.
2. If $\text{radix}$ is not present, let $\text{radixNumber}$ be 10.
3. Else if $\text{radix}$ is undefined, let $\text{radixNumber}$ be 10.
4. Else, let $\text{radixNumber}$ be $\text{ToInteger}(\text{radix})$.
5. If $\text{radixNumber} < 2$ or $\text{radixNumber} > 36$, throw a $\text{RangeError}$ exception.
6. If $\text{radixNumber} = 10$, return $\text{!ToString}(x)$.
7. Return the String representation of this Number value using the radix specified by $\text{radixNumber}$. Letters a-z are used for digits with values 10 through 35. The precise algorithm is implementation-dependent, however the algorithm should be a generalization of that specified in 7.1.12.1.

The $\text{toString}$ function is not generic; it throws a $\text{TypeError}$ 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 $\text{length}$ property of the $\text{toString}$ method is 1.

20.1.3.7 $\text{Number.prototype.valueOf()}$

1. Return $\text{? thisNumberValue(\text{this} \text{ value})}$.

20.1.4 Properties of Number Instances

Number instances are ordinary objects that inherit properties from the Number prototype object. Number instances also have a $[[\text{NumberData}]]$ internal slot. The $[[\text{NumberData}]]$ internal slot is the Number value represented by this Number object.

20.2 The Math Object

The Math object is the $\%\text{Math}\%$ intrinsic object and the initial value of the $\text{Math}$ property of the $\text{global object}$. The Math object is a single ordinary object.

The value of the $[[\text{Prototype}]]$ internal slot of the Math object is the intrinsic object $\%\text{ObjectPrototype}\%$.

The Math object is not a function object. It does not have a $[[\text{Construct}]]$ internal method; it is not possible to use the Math object as a constructor with the $\text{new}$ operator. The Math object also does not have a $[[\text{Call}]]$ internal method; it is not possible to invoke the Math object as a function.

NOTE In this specification, the phrase “the Number value for $x$” has a technical meaning defined in 6.1.6.
20.2.1 Value Properties of the Math Object

20.2.1.1 Math.E

The Number value for \( e \), the base of the natural logarithms, which is approximately 2.7182818284590452354.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

20.2.1.2 Math.LN10

The Number value for the natural logarithm of 10, which is approximately 2.302585092994046.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

20.2.1.3 Math.LN2

The Number value for the natural logarithm of 2, which is approximately 0.6931471805599453.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

20.2.1.4 Math.LOG10E

The Number value for the base-10 logarithm of \( e \), the base of the natural logarithms; this value is approximately 0.4342944819032518.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

| NOTE | The value of Math.LOG10E is approximately the reciprocal of the value of Math.LN10.

20.2.1.5 Math.LOG2E

The Number value for the base-2 logarithm of \( e \), the base of the natural logarithms; this value is approximately 1.4426950408889634.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

| NOTE | The value of Math.LOG2E is approximately the reciprocal of the value of Math.LN2.

20.2.1.6 Math.PI

The Number value for \( \pi \), the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.
This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \}.

20.2.1.7 Math.SQRT1_2

The Number value for the square root of \( \frac{1}{2} \), which is approximately 0.7071067811865476.

This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \}.

NOTE | The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.

20.2.1.8 Math.SQRT2

The Number value for the square root of 2, which is approximately 1.4142135623730951.

This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \}.

20.2.1.9 Math [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "Math".

This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true \}.

20.2.2 Function Properties of the Math Object

Each of the following Math object functions applies the ToNumber abstract operation to each of its arguments (in left-to-right order if there is more than one). If ToNumber returns an abrupt completion, that Completion Record is immediately returned. Otherwise, the function performs a computation on the resulting Number value(s). The value returned by each function is a Number.

In the function descriptions below, the symbols NaN, -0, +0, -\( \infty \) and +\( \infty \) refer to the Number values described in 6.1.6.

NOTE | The behaviour of the functions acos, acosh, asin, asinh, atan, atanh, atan2, cbrt, cos, cosh, exp, 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 754-2008 arithmetic contained in fdlibm, the freely distributable mathematical library from Sun Microsystems (http://www.netlib.org/fdlibm).

### 20.2.2.1 Math.abs (x)

Returns the absolute value of x; the result has the same magnitude as x but has positive sign.

- If \(x\) is **NaN**, the result is **NaN**.
- If \(x\) is -0, the result is +0.
- If \(x\) is -\(\infty\), the result is +\(\infty\).

### 20.2.2.2 Math.acos (x)

Returns an implementation-dependent approximation to the arc cosine of \(x\). The result is expressed in radians and ranges from +0 to +\(\pi\).

- If \(x\) is **NaN**, the result is **NaN**.
- If \(x\) is greater than 1, the result is **NaN**.
- If \(x\) is less than -1, the result is **NaN**.
- If \(x\) is exactly 1, the result is +0.

### 20.2.2.3 Math.acosh (x)

Returns an implementation-dependent approximation to the inverse hyperbolic cosine of \(x\).

- If \(x\) is **NaN**, the result is **NaN**.
- If \(x\) is less than 1, the result is **NaN**.
- If \(x\) is 1, the result is +0.
- If \(x\) is +\(\infty\), the result is +\(\infty\).

### 20.2.2.4 Math.asin (x)

Returns an implementation-dependent approximation to the arc sine of \(x\). The result is expressed in radians and ranges from -\(\pi/2\) to +\(\pi/2\).

- If \(x\) is **NaN**, the result is **NaN**.
- If \(x\) is greater than 1, the result is **NaN**.
- If \(x\) is less than -1, the result is **NaN**.
- If \(x\) is +0, the result is +0.
- If \(x\) is -0, the result is -0.
20.2.2.5 Math.asinh (x)

Returns an implementation-dependent approximation to the inverse hyperbolic sine of x.

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is -0, the result is -0.
- If \( x \) is +\( \infty \), the result is +\( \infty \).
- If \( x \) is -\( \infty \), the result is -\( \infty \).

20.2.2.6 Math.atan (x)

Returns an implementation-dependent approximation to the arc tangent of \( x \). The result is expressed in radians and ranges from -\( \pi / 2 \) to +\( \pi / 2 \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is -0, the result is -0.
- If \( x \) is +\( \infty \), the result is an implementation-dependent approximation to +\( \pi / 2 \).
- If \( x \) is -\( \infty \), the result is an implementation-dependent approximation to -\( \pi / 2 \).

20.2.2.7 Math.atanh (x)

Returns an implementation-dependent approximation to the inverse hyperbolic tangent of \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is less than -1, the result is NaN.
- If \( x \) is greater than 1, the result is NaN.
- If \( x \) is -1, the result is -\( \infty \).
- If \( x \) is +1, the result is +\( \infty \).
- If \( x \) is +0, the result is +0.
- If \( x \) is -0, the result is -0.

20.2.2.8 Math.atan2 (y, x)

Returns an implementation-dependent approximation to the arc tangent of the quotient \( y/x \) of the arguments \( y \) and \( x \), where the signs of \( y \) and \( x \) are used to determine the quadrant of the result. Note that it is intentional and traditional for the two-argument arc tangent function that the argument named \( y \) be first and the argument named \( x \) be second. The result is expressed in radians and ranges from -\( \pi \) to +\( \pi \).

- If either \( x \) or \( y \) is NaN, the result is NaN.
- If \( y > 0 \) and \( x \) is +0, the result is an implementation-dependent approximation to +\( \pi / 2 \).
- If \( y > 0 \) and \( x \) is -0, the result is an implementation-dependent approximation to +\( \pi / 2 \).
If $y$ is $+0$ and $x>0$, the result is $+0$.  
If $y$ is $+0$ and $x$ is $+0$, the result is $+0$.  
If $y$ is $+0$ and $x$ is $-0$, the result is an implementation-dependent approximation to $+\pi$.  
If $y$ is $+0$ and $x<0$, the result is an implementation-dependent approximation to $+\pi$.

If $y$ is $-0$ and $x>0$, the result is $-0$.  
If $y$ is $-0$ and $x$ is $+0$, the result is $-0$.  
If $y$ is $-0$ and $x$ is $-0$, the result is an implementation-dependent approximation to $-\pi$.  
If $y$ is $-0$ and $x<0$, the result is an implementation-dependent approximation to $-\pi$.  
If $y<0$ and $x$ is $+0$, the result is an implementation-dependent approximation to $-\pi/2$.  
If $y<0$ and $x$ is $-0$, the result is an implementation-dependent approximation to $-\pi/2$.

If $y>0$ and $y$ is finite and $x$ is $+\infty$, the result is $+0$.  
If $y>0$ and $y$ is finite and $x$ is $-\infty$, the result is an implementation-dependent approximation to $+\pi$.  
If $y<0$ and $y$ is finite and $x$ is $+\infty$, the result is $-0$.  
If $y<0$ and $y$ is finite and $x$ is $-\infty$, the result is an implementation-dependent approximation to $-\pi$.  
If $y$ is $+\infty$ and $x$ is finite, the result is an implementation-dependent approximation to $+\pi/2$.  
If $y$ is $-\infty$ and $x$ is finite, the result is an implementation-dependent approximation to $-\pi/2$.  
If $y$ is $+\infty$ and $x$ is $+\infty$, the result is an implementation-dependent approximation to $+\pi/4$.  
If $y$ is $+\infty$ and $x$ is $-\infty$, the result is an implementation-dependent approximation to $-3\pi/4$.  
If $y$ is $-\infty$ and $x$ is $+\infty$, the result is an implementation-dependent approximation to $-\pi/4$.  
If $y$ is $-\infty$ and $x$ is $-\infty$, the result is an implementation-dependent approximation to $-3\pi/4$.

### 20.2.2.9 Math.cbrt ( $x$ )

Returns an implementation-dependent approximation to the cube root of $x$.

- If $x$ is NaN, the result is NaN.  
- If $x$ is $+0$, the result is $+0$.  
- If $x$ is $-0$, the result is $-0$.  
- If $x$ is $+\infty$, the result is $+\infty$.  
- If $x$ is $-\infty$, the result is $-\infty$.

### 20.2.2.10 Math.ceil ( $x$ )

Returns the smallest (closest to $-\infty$) Number value that is not less than $x$ and is equal to a mathematical integer. If $x$ is already an integer, the result is $x$.

- If $x$ is NaN, the result is NaN.  
- If $x$ is $+0$, the result is $+0$.  
- If $x$ is $-0$, the result is $-0$.  
- If $x$ is $+\infty$, the result is $+\infty$.  
- If $x$ is $-\infty$, the result is $-\infty$.  

- If $x$ is less than 0 but greater than -1, the result is $-0$.

The value of $\text{Math.ceil}(x)$ is the same as the value of $-\text{Math.floor}(-x)$.

### 20.2.2.11 Math.clz32 ($x$)

When $\text{Math.clz32}$ is called with one argument $x$, the following steps are taken:

1. Let $n$ be $\text{ToUint32}(x)$.
2. Let $p$ be the number of leading zero bits in the 32-bit binary representation of $n$.

**NOTE** If $n$ is 0, $p$ will be 32. If the most significant bit of the 32-bit binary encoding of $n$ is 1, $p$ will be 0.

### 20.2.2.12 Math.cos ($x$)

Returns an implementation-dependent approximation to the cosine of $x$. The argument is expressed in radians.

- If $x$ is $\text{NaN}$, the result is $\text{NaN}$.
- If $x$ is $+0$, the result is 1.
- If $x$ is $-0$, the result is 1.
- If $x$ is $+\infty$, the result is $\text{NaN}$.
- If $x$ is $-\infty$, the result is $\text{NaN}$.

### 20.2.2.13 Math.cosh ($x$)

Returns an implementation-dependent approximation to the hyperbolic cosine of $x$.

- If $x$ is $\text{NaN}$, the result is $\text{NaN}$.
- If $x$ is $+0$, the result is 1.
- If $x$ is $-0$, the result is 1.
- If $x$ is $+\infty$, the result is $+\infty$.
- If $x$ is $-\infty$, the result is $+\infty$.

**NOTE** The value of $\text{cosh}(x)$ is the same as $(\exp(x) + \exp(-x))/2$.

### 20.2.2.14 Math.exp ($x$)

Returns an implementation-dependent approximation to the exponential function of $x$ ($e$ raised to the power of $x$, where $e$ is the base of the natural logarithms).
- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(+0\), the result is 1.
- If \( x \) is \(-0\), the result is 1.
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(+0\).

### 20.2.2.15 Math.expm1 \( (x) \)

Returns an implementation-dependent approximation to subtracting 1 from the exponential function of \( x \) (\( e \) raised to the power of \( x \), where \( e \) is the base of the natural logarithms). The result is computed in a way that is accurate even when the value of \( x \) is close 0.

- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-1\).

### 20.2.2.16 Math.floor \( (x) \)

Returns the greatest (closest to \( +\infty \)) Number value that is not greater than \( x \) and is equal to a mathematical integer. If \( x \) is already an integer, the result is \( x \).

- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-\infty\).
- If \( x \) is greater than 0 but less than 1, the result is \(+0\).

**NOTE**  The value of \( \text{Math.floor}(x) \) is the same as the value of \(-\text{Math.ceil}(-x)\).

### 20.2.2.17 Math.fround \( (x) \)

When \( \text{Math.fround} \) is called with argument \( x \), the following steps are taken:

1. If \( x \) is \( \text{NaN} \), return \( \text{NaN} \).
2. If \( x \) is one of \(+0\), \(-0\), \(+\infty\), \(-\infty\), return \( x \).
3. Let \( x_{32} \) be the result of converting \( x \) to a value in IEEE 754-2008 binary32 format using \( \text{roundTiesToEven} \).
4. Let \( x_{64} \) be the result of converting \( x_{32} \) to a value in IEEE 754-2008 binary64 format.
5. Return the ECMAScript Number value corresponding to \( x_{64} \).
20.2.2.18 Math.hypot (value1, value2, ...values)

**Math.hypot** returns an implementation-dependent approximation of the square root of the sum of squares of its arguments.

- If no arguments are passed, the result is +0.
- If any argument is +∞, the result is +∞.
- If any argument is -∞, the result is +∞.
- If no argument is +∞ or -∞, and any argument is NaN, the result is NaN.
- If all arguments are either +0 or -0, the result is +0.

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.

20.2.2.19 Math.imul (x, y)

When **Math.imul** is called with arguments x and y, the following steps are taken:

1. Let a be ToUInt32(x).
2. Let b be ToUInt32(y).
3. Let product be \((a \times b) \mod 2^{32}\).
4. If product ≥ \(2^{31}\), return product - \(2^{32}\); otherwise return product.

20.2.2.20 Math.log (x)

Returns an implementation-dependent approximation to the natural logarithm of x.

- If x is NaN, the result is NaN.
- If x is less than 0, the result is NaN.
- If x is +0 or -0, the result is -∞.
- If x is 1, the result is +0.
- If x is +∞, the result is +∞.

20.2.2.21 Math.log1p (x)

Returns an implementation-dependent approximation to the natural logarithm of \(1 + x\). The result is computed in a way that is accurate even when the value of x is close to zero.

- If x is NaN, the result is NaN.
- If x is less than -1, the result is NaN.
- If x is -1, the result is -∞.
- If x is +0, the result is +0.
• If \( x \) is \(-0\), the result is \(-0\).
• If \( x \) is \(+\infty\), the result is \(+\infty\).

20.2.2.22 Math.log10 ( \( x \) )

Returns an implementation-dependent approximation to the base 10 logarithm of \( x \).

• If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
• If \( x \) is less than 0, the result is \( \text{NaN} \).
• If \( x \) is \(+0\), the result is \(-\infty\).
• If \( x \) is \(-0\), the result is \(-\infty\).
• If \( x \) is 1, the result is \(+0\).
• If \( x \) is \(+\infty\), the result is \(+\infty\).

20.2.2.23 Math.log2 ( \( x \) )

Returns an implementation-dependent approximation to the base 2 logarithm of \( x \).

• If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
• If \( x \) is less than 0, the result is \( \text{NaN} \).
• If \( x \) is \(+0\), the result is \(-\infty\).
• If \( x \) is \(-0\), the result is \(-\infty\).
• If \( x \) is 1, the result is \(+0\).
• If \( x \) is \(+\infty\), the result is \(+\infty\).

20.2.2.24 Math.max ( \( value1, value2, \ldots, values \) )

Given zero or more arguments, calls ToNumber on each of the arguments and returns the largest of the resulting values.

• If no arguments are given, the result is \(-\infty\).
• If any value is \( \text{NaN} \), the result is \( \text{NaN} \).
• 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 \(-0\).

20.2.2.25 Math.min ( \( value1, value2, \ldots, values \) )

Given zero or more arguments, calls ToNumber on each of the arguments and returns the smallest of the resulting values.

• If no arguments are given, the result is \(+\infty\).
• If any value is \( \text{NaN} \), the result is \( \text{NaN} \).
• The comparison of values to determine the smallest value is done using the Abstract Relational Comparison algorithm except that $+0$ is considered to be larger than $-0$.

20.2.2.26 Math.pow (base, exponent)

1. Return the result of Applying the ** operator with base and exponent as specified in 12.6.4.

20.2.2.27 Math.random ()

Returns a Number value with positive sign, greater than or equal to 0 but less than 1, chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

Each Math.random function created for distinct realms must produce a distinct sequence of values from successive calls.

20.2.2.28 Math.round (x)

Returns the Number value that is closest to $x$ and is equal to a mathematical integer. If two integer Number values are equally close to $x$, then the result is the Number value that is closer to $+\infty$. If $x$ is already an integer, the result is $x$.

• If $x$ is NaN, the result is NaN.
• If $x$ is +0, the result is +0.
• If $x$ is -0, the result is -0.
• If $x$ is +\infty, the result is +\infty.
• If $x$ is -\infty, the result is -\infty.
• If $x$ is greater than 0 but less than 0.5, the result is +0.
• If $x$ is less than 0 but greater than or equal to -0.5, the result is -0.

| NOTE 1 | Math.round(3.5) returns 4, but Math.round(-3.5) returns -3. |
| NOTE 2 | The value of Math.round(x) is not always the same as the value of Math.floor(x+0.5). When $x$ is -0 or is less than 0 but greater than or equal to -0.5, Math.round(x) returns -0, but Math.floor(x+0.5) returns +0. Math.round(x) may also differ from the value of Math.floor(x+0.5) because of internal rounding when computing $x+0.5$. |

20.2.2.29 Math.sign (x)

Returns the sign of $x$, indicating whether $x$ is positive, negative, or zero.

• If $x$ is NaN, the result is NaN.
• If \(x\) is -0, the result is -0.
• If \(x\) is +0, the result is +0.
• If \(x\) is negative and not -0, the result is -1.
• If \(x\) is positive and not +0, the result is +1.

20.2.2.30 Math.sin (x)

Returns an implementation-dependent approximation to the sine of \(x\). The argument is expressed in radians.

- If \(x\) is NaN, the result is NaN.
- If \(x\) is +0, the result is +0.
- If \(x\) is -0, the result is -0.
- If \(x\) is +\(\infty\) or -\(\infty\), the result is NaN.

20.2.2.31 Math.sinh (x)

Returns an implementation-dependent approximation to the hyperbolic sine of \(x\).

- If \(x\) is NaN, the result is NaN.
- If \(x\) is +0, the result is +0.
- If \(x\) is -0, the result is -0.
- If \(x\) is +\(\infty\), the result is +\(\infty\).
- If \(x\) is -\(\infty\), the result is -\(\infty\).

**NOTE** The value of sinh(x) is the same as \((\exp(x) - \exp(-x))/2\).

20.2.2.32 Math.sqrt (x)

Returns an implementation-dependent approximation to the square root of \(x\).

- If \(x\) is NaN, the result is NaN.
- If \(x\) is less than 0, the result is NaN.
- If \(x\) is +0, the result is +0.
- If \(x\) is -0, the result is -0.
- If \(x\) is +\(\infty\), the result is +\(\infty\).

20.2.2.33 Math.tan (x)

Returns an implementation-dependent approximation to the tangent of \(x\). The argument is expressed in radians.
- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\) or \(-\infty\), the result is \( \text{NaN} \).

20.2.2.34 Math.tanh (\( x \))

Returns an implementation-dependent approximation to the hyperbolic tangent of \( x \).

- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+1\).
- If \( x \) is \(-\infty\), the result is \(-1\).

**NOTE**  The value of \( \tanh(x) \) is the same as \( \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \).

20.2.2.35 Math.trunc (\( x \))

Returns the integral part of the number \( x \), removing any fractional digits. If \( x \) is already an integer, the result is \( x \).

- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-\infty\).
- If \( x \) is greater than 0 but less than 1, the result is \(+0\).
- If \( x \) is less than 0 but greater than -1, the result is \(-0\).

20.3 Date Objects

20.3.1 Overview of Date Objects and Definitions of Abstract Operations

The following functions are abstract operations that operate on time values (defined in 20.3.1.1). Note that, in every case, if any argument to one of these functions is \( \text{NaN} \), the result will be \( \text{NaN} \).

20.3.1.1 Time Values and Time Range
A Date object contains a Number indicating a particular instant in time to within a millisecond. Such a Number is called a time value. A time value may also be NaN, indicating that the Date object does not represent a specific instant of time.

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. In time values leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript Number values can represent all integers from -9,007,199,254,740,992 to 9,007,199,254,740,992; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly -100,000,000 days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value +0.

### 20.3.1.2 Day Number and Time within Day

A given time value \( t \) belongs to day number

\[
\text{Day}(t) = \text{floor}(t / \text{msPerDay})
\]

where the number of milliseconds per day is

\[
\text{msPerDay} = 86400000
\]

The remainder is called the time within the day:

\[
\text{TimeWithinDay}(t) = t \text{ modulo msPerDay}
\]

### 20.3.1.3 Year Number

ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number \( y \) is therefore defined by

\[
\text{DaysInYear}(y)
\]

\[
= 365 \text{ if } (y \text{ modulo } 4) \neq 0
\]

\[
= 366 \text{ if } (y \text{ modulo } 4) = 0 \text{ and } (y \text{ modulo } 100) \neq 0
\]

\[
= 365 \text{ if } (y \text{ modulo } 100) = 0 \text{ and } (y \text{ modulo } 400) \neq 0
\]

\[
= 366 \text{ if } (y \text{ modulo } 400) = 0
\]
All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year \( y \) is given by:

\[
\text{DayFromYear}(y) = 365 \times (y-1970) + \text{floor}(y-1969)/4 - \text{floor}(y-1901)/100 + \text{floor}(y-1601)/400
\]

The time value of the start of a year is:

\[
\text{TimeFromYear}(y) = \text{msPerDay} \times \text{DayFromYear}(y)
\]

A time value determines a year by:

\[
\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that } \text{TimeFromYear}(y) \leq t
\]

The leap-year function is 1 for a time within a leap year and otherwise is zero:

\[
\text{InLeapYear}(t)
\begin{align*}
= 0 & \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 365 \\
= 1 & \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 366
\end{align*}
\]

### 20.3.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping \( \text{MonthFromTime}(t) \) from a time value \( t \) to a month number is defined by:

\[
\text{MonthFromTime}(t)
\begin{align*}
= 0 & \text{ if } 0 \leq \text{DayWithinYear}(t) < 31 \\
= 1 & \text{ if } 31 \leq \text{DayWithinYear}(t) < 59 + \text{InLeapYear}(t) \\
= 2 & \text{ if } 59 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90 + \text{InLeapYear}(t) \\
= 3 & \text{ if } 90 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120 + \text{InLeapYear}(t) \\
= 4 & \text{ if } 120 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151 + \text{InLeapYear}(t) \\
= 5 & \text{ if } 151 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181 + \text{InLeapYear}(t) \\
= 6 & \text{ if } 181 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212 + \text{InLeapYear}(t) \\
= 7 & \text{ if } 212 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243 + \text{InLeapYear}(t) \\
= 8 & \text{ if } 243 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273 + \text{InLeapYear}(t) \\
= 9 & \text{ if } 273 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304 + \text{InLeapYear}(t) \\
= 10 & \text{ if } 304 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334 + \text{InLeapYear}(t) \\
= 11 & \text{ if } 334 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365 + \text{InLeapYear}(t)
\end{align*}
\]

where

\[
\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))
\]

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10
specifies November; and 11 specifies December. Note that MonthFromTime(0) = 0, corresponding to Thursday, 01 January, 1970.

### 20.3.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping DateFromTime(t) from a time value t to a date number is defined by:

\[
\text{DateFromTime}(t) =
\begin{align*}
&\text{DayWithinYear}(t)+1 \text{ if } \text{MonthFromTime}(t)=0 \\
&\text{DayWithinYear}(t)-30 \text{ if } \text{MonthFromTime}(t)=1 \\
&\text{DayWithinYear}(t)-58 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=2 \\
&\text{DayWithinYear}(t)-89 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=3 \\
&\text{DayWithinYear}(t)-119 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=4 \\
&\text{DayWithinYear}(t)-150 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=5 \\
&\text{DayWithinYear}(t)-180 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=6 \\
&\text{DayWithinYear}(t)-211 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=7 \\
&\text{DayWithinYear}(t)-242 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=8 \\
&\text{DayWithinYear}(t)-272 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=9 \\
&\text{DayWithinYear}(t)-303 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=10 \\
&\text{DayWithinYear}(t)-333 - \text{InLeapYear}(t) \text{ if } \text{MonthFromTime}(t)=11
\end{align*}
\]

### 20.3.1.6 Week Day

The weekday for a particular time value t is defined as

\[
\text{WeekDay}(t) = (\text{Day}(t) + 4) \text{ modulo } 7
\]

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that WeekDay(0) = 4, corresponding to Thursday, 01 January, 1970.

### 20.3.1.7 Local Time Zone Adjustment

An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value LocalTZA measured in milliseconds which when added to UTC represents the local standard time. Daylight saving time is not reflected by LocalTZA.

**NOTE** It is recommended that implementations use the time zone information of the IANA Time Zone Database [https://www.iana.org/time-zones/](https://www.iana.org/time-zones/).

### 20.3.1.8 Daylight Saving Time Adjustment
An implementation-dependent algorithm using best available information on time zones to determine the local daylight saving time adjustment \(\text{DaylightSavingTA}(t)\), measured in milliseconds. An implementation of ECMAScript is expected to make its best effort to determine the local daylight saving time adjustment.

**NOTE**
It is recommended that implementations use the time zone information of the IANA Time Zone Database [https://www.iana.org/time-zones/](https://www.iana.org/time-zones/).

### 20.3.1.9 LocalTime (\( t \))

The abstract operation \(\text{LocalTime}(t)\) converts \( t \) from UTC to local time by performing the following steps:

1. Return \( t + \text{LocalTZA} + \text{DaylightSavingTA}(t) \).

### 20.3.1.10 UTC (\( t \))

The abstract operation \(\text{UTC}(t)\) converts \( t \) from local time to UTC. It performs the following steps:

1. Return \( t - \text{LocalTZA} - \text{DaylightSavingTA}(t - \text{LocalTZA}) \).

**NOTE**
\(\text{UTC(\text{LocalTime}(t))}\) is not necessarily always equal to \( t \).

### 20.3.1.11 Hours, Minutes, Second, and Milliseconds

The following abstract operations are useful in decomposing time values:

\[
\begin{align*}
\text{HourFromTime}(t) &= \text{floor}(t / \text{msPerHour}) \mod \text{HoursPerDay} \\
\text{MinFromTime}(t) &= \text{floor}(t / \text{msPerMinute}) \mod \text{MinutesPerHour} \\
\text{SecFromTime}(t) &= \text{floor}(t / \text{msPerSecond}) \mod \text{SecondsPerMinute} \\
\text{msFromTime}(t) &= t \mod \text{msPerSecond}
\end{align*}
\]

where

\[
\begin{align*}
\text{HoursPerDay} &= 24 \\
\text{MinutesPerHour} &= 60 \\
\text{SecondsPerMinute} &= 60 \\
\text{msPerSecond} &= 1000 \\
\text{msPerMinute} &= \text{60000} = \text{msPerSecond} \times \text{SecondsPerMinute} \\
\text{msPerHour} &= \text{3600000} = \text{msPerMinute} \times \text{MinutesPerHour}
\end{align*}
\]

### 20.3.1.12 MakeTime (\( hour, min, sec, ms \))
The abstract operation MakeTime calculates a number of milliseconds from its four arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `hour` is not finite or `min` is not finite or `sec` is not finite or `ms` is not finite, return `NaN.`
2. Let `h` be `!ToInteger(hour).`
3. Let `m` be `!ToInteger(min).`
4. Let `s` be `!ToInteger(sec).`
5. Let `milli` be `!ToInteger(ms).`
6. Let `t` be `h * msPerHour + m * msPerMinute + s * msPerSecond + milli`, performing the arithmetic according to IEEE 754-2008 rules (that is, as if using the ECMAScript operators * and +).  
7. Return `t`.

### 20.3.1.13 MakeDay (`year, month, date`)

The abstract operation MakeDay calculates a number of days from its three arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `year` is not finite or `month` is not finite or `date` is not finite, return `NaN.`
2. Let `y` be `!ToInteger(year).`
3. Let `m` be `!ToInteger(month).`
4. Let `dt` be `!ToInteger(date).`
5. Let `ym` be `y + floor(m / 12).`
6. Let `mn` be `m modulo 12.`
7. Find a value `t` such that `YearFromTime(t)` is `ym` and `MonthFromTime(t)` is `mn` and `DateFromTime(t)` is `1`; but if this is not possible (because some argument is out of range), return `NaN.`

### 20.3.1.14 MakeDate (`day, time`)

The abstract operation MakeDate calculates a number of milliseconds from its two arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `day` is not finite or `time` is not finite, return `NaN.`
2. Return `day * msPerDay + time.`

### 20.3.1.15 TimeClip (`time`)

The abstract operation TimeClip calculates a number of milliseconds from its argument, which must be an ECMAScript Number value. This operator functions as follows:

1. If `time` is not finite, return `NaN.`
2. If `abs(time) > 8.64 × 10^{15}`, return `NaN.`
3. Let \textit{clippedTime} be \( \text{ToInteger}(\text{time}) \).
4. If \textit{clippedTime} is \(-0\), set \textit{clippedTime} to \(+0\).
5. Return \textit{clippedTime}.

\textbf{NOTE} The point of step 4 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish \(-0\) and \(+0\).

\subsection*{20.3.1.16 Date Time String Format}

ECMAScript defines a string interchange format for date-times based upon a simplification of the ISO 8601 Extended Format. The format is as follows: \texttt{YYYY-MM-DDTHH:mm:ss.sssZ}

Where the fields are as follows:

- \texttt{YYYY} is the decimal digits of the year 0000 to 9999 in the Gregorian calendar.
- \texttt{-} "-" (hyphen) appears literally twice in the string.
- \texttt{MM} is the month of the year from 01 (January) to 12 (December).
- \texttt{DD} is the day of the month from 01 to 31.
- \texttt{T} "T" appears literally in the string, to indicate the beginning of the time element.
- \texttt{HH} is the number of complete hours that have passed since midnight as two decimal digits from 00 to 24.
- \texttt{:} "::" (colon) appears literally twice in the string.
- \texttt{mm} is the number of complete minutes since the start of the hour as two decimal digits from 00 to 59.
- \texttt{ss} is the number of complete seconds since the start of the minute as two decimal digits from 00 to 59.
- \texttt{.} "." (dot) appears literally in the string.
- \texttt{sss} is the number of complete milliseconds since the start of the second as three decimal digits.
- \texttt{Z} is the time zone offset specified as "Z" (for UTC) or either "+" or "-" followed by a time expression \texttt{HH:mm}

This format includes date-only forms:

- \texttt{YYYY}
- \texttt{YYYY-MM}
- \texttt{YYYY-MM-DD}
It also includes “date-time” forms that consist of one of the above date-only forms immediately followed by one of the following time forms with an optional time zone offset appended:

THH:mm
THH:mm:ss
THH:mm:ss.sss

All numbers must be base 10. If the MM or DD fields are absent "01" is used as the value. If the HH, mm, or ss fields are absent "00" is used as the value and the value of an absent sss field is "000". When the time zone offset is absent, date-only forms are interpreted as a UTC time and date-time forms are interpreted as a local time.

Illegal values (out-of-bounds as well as syntax errors) in a format string means that the format string is not a valid instance of this format.

NOTE 1  As every day both starts and ends with midnight, the two notations 00:00 and 24:00 are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: 1995-02-04T24:00 and 1995-02-05T00:00

NOTE 2  There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. For this reason, ISO 8601 and this format specifies numeric representations of date and time.

20.3.1.16.1 Extended Years

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,426 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the simplified ECMAScript format such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or - sign. The year 0 is considered positive and hence prefixed with a + sign.

NOTE  Examples of extended years:

-283457-03-21T15:00:59.008Z  283458 B.C.
-000001-01-01T00:00:00Z  2 B.C.
+000000-01-01T00:00:00Z  1 B.C.
+000001-01-01T00:00:00Z  1 A.D.
+001970-01-01T00:00:00Z  1970 A.D.
+002009-12-15T00:00:00Z  2009 A.D.
20.3.2 The Date Constructor

The Date constructor is the `%Date% intrinsic object and the initial value of the Date property of the global object. When called as a constructor it creates and initializes a new Date object. When Date is called as a function rather than as a constructor, it returns a String representing the current time (UTC).

The Date constructor is a single function whose behaviour is overloaded based upon the number and types of its arguments.

The Date 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 Date behaviour must include a super call to the Date constructor to create and initialize the subclass instance with a [[DateValue]] internal slot.

The length property of the Date constructor function is 7.

20.3.2.1 Date (year, month [, date [, hours [, minutes [, seconds [, ms ] ] ] ] ] )

This description applies only if the Date constructor is called with at least two arguments.

When the Date function is called, the following steps are taken:

1. Let numberOfArgs be the number of arguments passed to this function call.
3. If NewTarget is not undefined, then
   a. Let y be ? ToNumber(year).
   b. Let m be ? ToNumber(month).
   c. If date is supplied, let dt be ? ToNumber(date); else let dt be 1.
   d. If hours is supplied, let h be ? ToNumber(hours); else let h be 0.
   e. If minutes is supplied, let min be ? ToNumber(minutes); else let min be 0.
   f. If seconds is supplied, let s be ? ToNumber(seconds); else let s be 0.
   g. If ms is supplied, let milli be ? ToNumber(ms); else let milli be 0.
   h. If y is not NaN and 0 ≤ ToInteger(y) ≤ 99, let yr be 1900+ToInteger(y); otherwise, let yr be y.
   i. Let finalDate be MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli)).
   j. Let O be ? OrdinaryCreateFromConstructor(NewTarget, "%DatePrototype%", « [[DateValue]] »).
   k. Set O.[[DateValue]] to TimeClip(UTC(finalDate)).
   l. Return O.
4. Else,
   a. Let now be the Number that is the time value (UTC) identifying the current time.
   b. Return ToDateString(now).

20.3.2.2 Date ( value )

This description applies only if the Date constructor is called with exactly one argument.

When the Date function is called, the following steps are taken:

1. Let numberOfArgs be the number of arguments passed to this function call.
3. If NewTarget is not undefined, then
   a. If Type(value) is Object and value has a [[DateValue]] internal slot, then
      i. Let tv be thisTimeValue(value).
   b. Else,
      i. Let v be ? ToPrimitive(value).
      ii. If Type(v) is String, then
          1. Let tv be the result of parsing v as a date, in exactly the same manner as for the
             parse method (20.3.2.2). If the parse resulted in an abrupt completion, tv is the
             Completion Record.
          2. Return IfAbrupt(tv).
      iii. Else,
          1. Let tv be ? ToNumber(v).
   c. Let O be ? OrdinaryCreateFromConstructor(NewTarget, "%DatePrototype", « [[DateValue]] »).
   d. Set O.[[DateValue]] to TimeClip(tv).
   e. Return O.
4. Else,
   a. Let now be the Number that is the time value (UTC) identifying the current time.
   b. Return ToDateString(now).

20.3.2.3 Date ( )

This description applies only if the Date constructor is called with no arguments.

When the Date function is called, the following steps are taken:

1. Let numberOfArgs be the number of arguments passed to this function call.
2. Assert: numberOfArgs = 0.
3. If NewTarget is not undefined, then
a. Let \( O \) be \( \text{OrdinaryCreateFromConstructor} \( \text{NewTarget, "%DatePrototype%", « [[DateValue]] »} \).  

b. Set \( O.\text{[DateValue]} \) to the \textit{time value} (UTC) identifying the current time.  
c. Return \( O \).

4. Else,  
a. Let \( \text{now} \) be the Number that is the \textit{time value} (UTC) identifying the current time.  
b. Return \( \text{ToDate} \text{String}(\text{now}) \).

### 20.3.3 Properties of the Date Constructor

The value of the \([\text{Prototype}]\) internal slot of the Date constructor is the intrinsic object \%FunctionPrototype\%.

The Date constructor has the following properties:

#### 20.3.3.1 Date.now ( )

The \textit{now} function returns a Number value that is the \textit{time value} designating the UTC date and time of the occurrence of the call to \textit{now}.

#### 20.3.3.2 Date.parse ( string )

The \textit{parse} function applies the \textit{ToString} operator to its argument. If \textit{ToString} results in an \textit{abrupt completion} the \textit{Completion Record} is immediately returned. Otherwise, \textit{parse} interprets the resulting String as a date and time; it returns a Number, the UTC \textit{time value} corresponding to the date and time. The String may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the String. The function first attempts to parse the format of the String according to the rules (including extended years) called out in Date Time String Format (20.3.1.16). If the String does not conform to that format the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Unrecognizable Strings or dates containing illegal element values in the format String shall cause \textbf{Date.parse} to return \textbf{NaN}.

If \( x \) is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

\[
x.\text{valueOf()}
\]

\[
\text{Date.parse(x.toString())}
\]

\[
\text{Date.parse(x.toUTCString())}
\]

\[
\text{Date.parse(x.toISOString())}
\]

However, the expression
Date.parse(x.toLocaleString())

is not required to produce the same Number value as the preceding three expressions and, in general, the value produced by Date.parse is implementation-dependent when given any String value that does not conform to the Date Time String Format (20.3.1.16) and that could not be produced in that implementation by the toString or toUTCString method.

20.3.3.3 Date.prototype

The initial value of Date.prototype is the intrinsic object %DatePrototype%. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false.

20.3.4 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 supplied, let m be ? ToNumber(month); else let m be 0.
3. If date is supplied, let dt be ? ToNumber(date); else let dt be 1.
4. If hours is supplied, let h be ? ToNumber(hours); else let h be 0.
5. If minutes is supplied, let min be ? ToNumber(minutes); else let min be 0.
6. If seconds is supplied, let s be ? ToNumber(seconds); else let s be 0.
7. If ms is supplied, let milli be ? ToNumber(ms); else let milli be 0.
8. If y is not NaN and 0 ≤ ToInteger(y) ≤ 99, let yr be 1900+ToInteger(y); otherwise, let yr be y.
9. Return TimeClip(MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli))).

The length property of the UTC function is 7.

NOTE The UTC function differs from the Date constructor in two ways: it returns a time value as a Number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

20.3.4 Properties of the Date Prototype Object

The Date prototype object is the intrinsic object %DatePrototype%. The Date prototype object is itself an ordinary object. It is not a Date instance and does not have a [[DateValue]] internal slot.

The value of the [[Prototype]] internal slot of the Date prototype object is the intrinsic object %ObjectPrototype%.

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(value)` performs the following steps:

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.

### 20.3.4.1 Date.prototype.constructor

The initial value of `Date.prototype.constructor` is the intrinsic object `%Date%`.

### 20.3.4.2 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))`.

### 20.3.4.3 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))`.

### 20.3.4.4 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))`.

### 20.3.4.5 Date.prototype.getHours ()
The following steps are performed:

1. Let \( t \) be ?\( \text{thisTimeValue} \) (this value).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{HourFromTime} \) (\( \text{LocalTime}(t) \)).

20.3.4.6 Date.prototype.getMilliseconds ()

The following steps are performed:

1. Let \( t \) be ?\( \text{thisTimeValue} \) (this value).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{msFromTime} \) (\( \text{LocalTime}(t) \)).

20.3.4.7 Date.prototype.getMinutes ()

The following steps are performed:

1. Let \( t \) be ?\( \text{thisTimeValue} \) (this value).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MinFromTime} \) (\( \text{LocalTime}(t) \)).

20.3.4.8 Date.prototype.getMonth ()

The following steps are performed:

1. Let \( t \) be ?\( \text{thisTimeValue} \) (this value).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MonthFromTime} \) (\( \text{LocalTime}(t) \)).

20.3.4.9 Date.prototype.getSeconds ()

The following steps are performed:

1. Let \( t \) be ?\( \text{thisTimeValue} \) (this value).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{SecFromTime} \) (\( \text{LocalTime}(t) \)).

20.3.4.10 Date.prototype.getTime ()

The following steps are performed:

1. Return ?\( \text{thisTimeValue} \) (this value).
The following steps are performed:

1. Let $t$ be ? thisTimeValue(this value).
2. If $t$ is NaN, return NaN.
3. Return $(t - \text{LocalTime}(t)) / \text{msPerMinute}$.

The following steps are performed:

1. Let $t$ be ? thisTimeValue(this value).
2. If $t$ is NaN, return NaN.
3. Return DateFromTime($t$).

The following steps are performed:

1. Let $t$ be ? thisTimeValue(this value).
2. If $t$ is NaN, return NaN.
3. Return WeekDay($t$).

The following steps are performed:

1. Let $t$ be ? thisTimeValue(this value).
2. If $t$ is NaN, return NaN.
3. Return YearFromTime($t$).

The following steps are performed:

1. Let $t$ be ? thisTimeValue(this value).
2. If $t$ is NaN, return NaN.
3. Return HourFromTime($t$).

The following steps are performed:
1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{msFromTime}(t) \).

20.3.4.17 \( \text{Date.prototype.getUTCMinutes}() \)

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MinFromTime}(t) \).

20.3.4.18 \( \text{Date.prototype.getUTCMonth}() \)

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{MonthFromTime}(t) \).

20.3.4.19 \( \text{Date.prototype.getUTCSecsconds}() \)

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
3. Return \( \text{SecFromTime}(t) \).

20.3.4.20 \( \text{Date.prototype.setDate} (\text{date}) \)

The following steps are performed:

1. Let \( t \) be \( \text{LocalTime}(\text{thisTimeValue}(\text{this value})) \).
2. Let \( dt \) be \( \text{ToNumber}(\text{date}) \).
3. Let \( \text{newDate} \) be \( \text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), \text{MonthFromTime}(t), dt), \text{TimeWithinDay}(t)) \).
4. Let \( u \) be \( \text{TimeClip}(\text{UTC}(\text{newDate})) \).
5. Set the \([\text{DateValue}]\) internal slot of this \text{Date} object to \( u \).
6. Return \( u \).

20.3.4.21 \( \text{Date.prototype.setFullYear} (\text{year}[, \text{month}[,,\text{date}]]) \)

The following steps are performed:
1. Let \( t \) be \(\text{thisTimeValue(this value)}\).
2. If \( t \) is \(\text{NaN} \), let \( t \) be \(+0\); otherwise, let \( t \) be \(\text{LocalTime(t)}\).
3. Let \( y \) be \(\text{ToNumber(year)}\).
4. If \( \text{month} \) is not specified, let \( m \) be \(\text{MonthFromTime(t)}\); otherwise, let \( m \) be \(\text{ToNumber(month)}\).
5. If \( \text{date} \) is not specified, let \( dt \) be \(\text{DateFromTime(t)}\); otherwise, let \( dt \) be \(\text{ToNumber(date)}\).
6. Let \( \text{newDate} \) be \(\text{MakeDate(MakeDay(y, m, dt), TimeWithinDay(t)})\).
7. Let \( u \) be \(\text{TimeClip(UTC(newDate))}\).
8. Set the [[DateValue]] internal slot of this Date object to \( u \).
9. Return \( u \).

The \textbf{length} property of the \texttt{setFullYear} method is 3.

\begin{itemize}
\item \textbf{NOTE} If \( \text{month} \) is not specified, this method behaves as if \( \text{month} \) were specified with the value \(\text{getMonth()}\). If \( \text{date} \) is not specified, it behaves as if \( \text{date} \) were specified with the value \(\text{getDate()}\).
\end{itemize}

\section*{20.3.4.22 Date.prototype.setHours( \texttt{hour \[ , min \[ , sec \[ , ms \] \] \]} \)}

The following steps are performed:

1. Let \( t \) be \(\text{LocalTime(? thisTimeValue(this value))}\).
2. Let \( h \) be \(\text{ToNumber(hour)}\).
3. If \( \text{min} \) is not specified, let \( m \) be \(\text{MinFromTime(t)}\); otherwise, let \( m \) be \(\text{ToNumber(min)}\).
4. If \( \text{sec} \) is not specified, let \( s \) be \(\text{SecFromTime(t)}\); otherwise, let \( s \) be \(\text{ToNumber(sec)}\).
5. If \( \text{ms} \) is not specified, let \( \text{milli} \) be \(\text{msFromTime(t)}\); otherwise, let \( \text{milli} \) be \(\text{ToNumber(ms)}\).
6. Let \( \text{date} \) be \(\text{MakeDate(Day(t), MakeTime(h, m, s, milli)})\).
7. Let \( u \) be \(\text{TimeClip(UTC(date))}\).
8. Set the [[DateValue]] internal slot of this Date object to \( u \).
9. Return \( u \).

The \textbf{length} property of the \texttt{setHours} method is 4.

\begin{itemize}
\item \textbf{NOTE} If \( \text{min} \) is not specified, this method behaves as if \( \text{min} \) were specified with the value \(\text{getMinutes()}\). If \( \text{sec} \) is not specified, it behaves as if \( \text{sec} \) were specified with the value \(\text{getSeconds()}\). If \( \text{ms} \) is not specified, it behaves as if \( \text{ms} \) were specified with the value \(\text{getMilliseconds()}\).
\end{itemize}

\section*{20.3.4.23 Date.prototype.setMilliseconds( \texttt{ms} \)}

The following steps are performed:

1. Let \( t \) be \(\text{LocalTime(? thisTimeValue(this value))}\).
2. Let \( \text{ms} \) be \(\text{ToNumber(ms)}\).
3. Let time be $\text{MakeTime(}\text{HourFromTime}(t), \text{MinFromTime}(t), \text{SecFromTime}(t), \text{ms})$.
4. Let $u$ be $\text{TimeClip(UTC(\text{MakeDate(Day}(t), \text{time}))}$.
5. Set the [[DateValue]] internal slot of this Date object to $u$.
6. Return $u$.

20.3.4.24 Date.prototype.setMinutes ($\text{min [ , sec [ , ms ] ]}$)

The following steps are performed:

1. Let $t$ be $\text{LocalTime(}?$ thisTimeValue(this value))$.
2. Let $m$ be $\text{? ToNumber(min)}$.
3. If sec is not specified, let $s$ be $\text{SecFromTime}(t)$; otherwise, let $s$ be $\text{? ToNumber(sec)}$.
4. If ms is not specified, let $milli$ be $\text{msFromTime(t)}$; otherwise, let $milli$ be $\text{? ToNumber(ms)}$.
5. Let $\text{date}$ be $\text{MakeDate(Day}(t), \text{MakeTime(HourFromTime}(t), m, s, milli))$.
6. Let $u$ be $\text{TimeClip(UTC(date))}$.
7. Set the [[DateValue]] internal slot of this Date object to $u$.
8. Return $u$.

The length property of the setMinutes method is 3.

NOTE If sec is not specified, this method behaves as if sec were specified with the value $\text{getSeconds()}$. If ms is not specified, this behaves as if ms were specified with the value $\text{getMilliseconds()}$.

20.3.4.25 Date.prototype.setMonth ($\text{month [ , date ]}$)

The following steps are performed:

1. Let $t$ be $\text{LocalTime(}?$ thisTimeValue(this value))$.
2. Let $m$ be $\text{? ToNumber(month)}$.
3. If date is not specified, let $dt$ be $\text{DateFromTime}(t)$; otherwise, let $dt$ be $\text{? ToNumber(date)}$.
4. Let $\text{newDate}$ be $\text{MakeDate(MakeDay(YearFromTime}(t), m, dt), TimeWithinDay(t)}$.
5. Let $u$ be $\text{TimeClip(UTC(newDate))}$.
6. Set the [[DateValue]] internal slot of this Date object to $u$.
7. Return $u$.

The length property of the setMonth method is 2.

NOTE If date is not specified, this method behaves as if date were specified with the value $\text{getDate()}$.

20.3.4.26 Date.prototype.setSeconds ($\text{sec [ , ms ]}$)
The following steps are performed:

1. Let \( t \) be \( \text{LocalTime}(? \thisTimeValue(\text{this value})) \).
2. Let \( s \) be \( ? \ToNumber(\text{sec}) \).
3. If \( ms \) is not specified, let \( milli \) be \( ms\FromTime(t) \); otherwise, let \( milli \) be \( ? \ToNumber(\text{ms}) \).
4. Let \( date \) be \( \MakeDate(\Day(t), \MakeTime(\HourFromTime(t), \MinFromTime(t), s, milli)) \).
5. Let \( u \) be \( \TimeClip(\UTC(date)) \).
6. Set the \([[\text{DateValue}]]\) internal slot of \( \text{this Date object} \) to \( u \).
7. Return \( u \).

The \texttt{length} property of the \texttt{setSeconds} method is 2.

**NOTE** If \( ms \) is not specified, this method behaves as if \( ms \) were specified with the value \( \text{getMilliseconds()} \).

### 20.3.4.27 \texttt{Date.prototype.setTime()} (\textit{time})

The following steps are performed:

1. Perform \( ? \thisTimeValue(\text{this value}) \).
2. Let \( t \) be \( ? \ToNumber(\text{time}) \).
3. Let \( v \) be \( \TimeClip(t) \).
4. Set the \([[\text{DateValue}]]\) internal slot of \( \text{this Date object} \) to \( v \).
5. Return \( v \).

### 20.3.4.28 \texttt{Date.prototype.setUTCDate()} (\textit{date})

1. Let \( t \) be \( ? \thisTimeValue(\text{this value}) \).
2. Let \( dt \) be \( ? \ToNumber(\text{date}) \).
3. Let \( \newDate \) be \( \MakeDate(\MakeDay(\YearFromTime(t), \MonthFromTime(t), dt), \TimeWithinDay(t)) \).
4. Let \( v \) be \( \TimeClip(\newDate) \).
5. Set the \([[\text{DateValue}]]\) internal slot of \( \text{this Date object} \) to \( v \).
6. Return \( v \).

### 20.3.4.29 \texttt{Date.prototype.setUTCFullYear()} (\textit{year} [ , \textit{month} [ , \textit{date} ]] )

The following steps are performed:

1. Let \( t \) be \( ? \thisTimeValue(\text{this value}) \).
2. If \( t \) is \( \text{NaN} \), let \( t \) be \( +0 \).
3. Let \( y \) be \( ? \ToNumber(\text{year}) \).
4. If \( \text{month} \) is not specified, let \( m \) be \( \MonthFromTime(t) \); otherwise, let \( m \) be \( ? \ToNumber(\text{month}) \).
5. If date is not specified, 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 specified, this method behaves as if month were specified with the value getUTCMonth(). If date is not specified, it behaves as if date were specified with the value getUTCDate().

20.3.4.30 Date.prototype.setUTCHours ( hour [, min [, sec [, ms ] ]] )

The following steps are performed:

1. Let t be ? thisTimeValue(this value).
2. Let h be ? ToNumber(hour).
3. If min is not specified, let m be MinFromTime(t); otherwise, let m be ? ToNumber(min).
4. If sec is not specified, let s be SecFromTime(t); otherwise, let s be ? ToNumber(sec).
5. If ms is not specified, 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 specified, this method behaves as if min were specified with the value getUTCMinutes(). If sec is not specified, it behaves as if sec were specified with the value getUTCSeconds(). If ms is not specified, it behaves as if ms were specified with the value getUTCMilliseconds().

20.3.4.31 Date.prototype.setUTCSeconds ( 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.
20.3.4.32 Date.prototype.setUTCMinutes ( \( min [, \ sec [, \ ms ] ] \) )

The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue} (\text{this value}) \).
2. Let \( m \) be \( \text{ToNumber} (\text{min}) \).
3. If \( \sec \) is not specified, let \( s \) be \( \text{SecFromTime} (t) \).
4. Else,
   a. Let \( s \) be \( \text{ToNumber} (\text{sec}) \).
5. If \( \ms \) is not specified, let \( \text{milli} \) be \( \text{msFromTime} (t) \).
6. Else,
   a. Let \( \text{milli} \) be \( \text{ToNumber} (\text{ms}) \).
7. Let \( date \) 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 \text{this Date object} to \( v \).
10. Return \( v \).

The \texttt{length} property of the \texttt{setUTCMinutes} method is 3.

\textbf{NOTE} If \( \sec \) is not specified, this method behaves as if \( \sec \) were specified with the value \text{getUTCSeconds}(). If \( \ms \) is not specified, it function behaves as if \( \ms \) were specified with the value return by \text{getUTCMilliseconds}().

20.3.4.33 Date.prototype.setUTCMonth ( \( \texttt{month} [, \ date ] \) )

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 specified, 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 \text{this Date object} to \( v \).
8. Return \( v \).

The \texttt{length} property of the \texttt{setUTCMonth} method is 2.

\textbf{NOTE} If \( \text{date} \) is not specified, this method behaves as if \( \text{date} \) were specified with the value \text{getUTCDate}().

20.3.4.34 Date.prototype.setUTCSeconds ( \( \texttt{sec} [, \ ms ] \) )
The following steps are performed:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this value}) \).
2. Let \( s \) be \( \text{ToNumber}(\text{sec}) \).
3. If \( ms \) is not specified, let \( milli \) be \( \text{msFromTime}(t) \).
4. Else,
   a. Let \( milli \) be \( \text{ToNumber}(ms) \).
5. Let \( date \) be \( \text{MakeDate}(\text{Day}(t), \text{MakeTime}(\text{HourFromTime}(t), \text{MinFromTime}(t), s, milli)) \).
6. Let \( v \) be \( \text{TimeClip}(date) \).
7. Set the \([[\text{DateValue}]]\) internal slot of \( \text{this Date object} \) to \( v \).
8. Return \( v \).

The \texttt{length} property of the \texttt{setUTCSecsonds} method is 2.

**NOTE**
If \( ms \) is not specified, this method behaves as if \( ms \) were specified with the value \( \text{getUTCMilliseconds()} \).

### 20.3.4.35 Date.prototype.toDateString()

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the “date” portion of the Date in the current time zone in a convenient, human-readable form.

### 20.3.4.36 Date.prototype.toISOString()

This function returns a String value representing the instance in time corresponding to \( \text{this time value} \). The format of the String is the Date Time string format defined in 20.3.1.16. All fields are present in the String. The time zone is always UTC, denoted by the suffix \( Z \). If \( \text{this time value} \) is not a finite Number or if the year is not a value that can be represented in that format (if necessary using extended year format), a \texttt{RangeError} exception is thrown.

### 20.3.4.37 Date.prototype.toJSON( key )

This function provides a String representation of a Date object for use by \texttt{JSON.stringify} (24.5.2).

When the \texttt{toJSON} method is called with argument \( key \), the following steps are taken:

1. Let \( O \) be \( \text{ToObject}(\text{this value}) \).
2. Let \( tv \) be \( \text{ToPrimitive}(O, \text{hint Number}) \).
3. If \( \text{Type}(tv) \) is Number and \( tv \) is not finite, return \texttt{null}.
4. Return \( ? \text{Invoke}(O, "toISOString") \).

**NOTE 1**
The argument is ignored.
The `JSON` 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 `toString` method.

20.3.4.38 `Date.prototype.toLocaleDateString` ([`reserved1`, `reserved2`])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the `Date.prototype.toLocaleDateString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleDateString` method is used.

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the “date” portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment’s current locale.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter positions for anything else.

20.3.4.39 `Date.prototype.toLocaleString` ([`reserved1`, `reserved2`])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the `Date.prototype.toLocaleString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleString` method is used.

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment’s current locale.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter positions for anything else.

20.3.4.40 `Date.prototype.toLocaleTimeString` ([`reserved1`, `reserved2`])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the `Date.prototype.toLocaleTimeString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleTimeString` method is used.
This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the “time” portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter positions for anything else.

### 20.3.4.41 Date.prototype.toString ()

The following steps are performed:

1. Let \( O \) be this Date object.
2. If \( O \) does not have a \([\text{DateValue}]\) internal slot, then
   a. Let \( tv \) be NaN.
3. Else,
   a. Let \( tv \) be thisTimeValue \((O)\).
4. Return ToDateString \((tv)\).

**NOTE 1**
For any Date object \( d \) whose milliseconds amount is zero, the result of `Date.parse(d.toString())` is equal to `d.valueOf()`. See 20.3.3.2.

**NOTE 2**
The `toString` 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.

### 20.3.4.41.1 Runtime Semantics: ToDateString \((tv)\)

The following steps are performed:

1. Assert: Type \((tv)\) is Number.
2. If \( tv \) is NaN, return "Invalid Date".
3. Return an implementation-dependent String value that represents \( tv \) as a date and time in the current time zone using a convenient, human-readable form.

### 20.3.4.42 Date.prototype.toTimeString ()

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the “time” portion of the Date in the current time zone in a convenient, human-readable form.

### 20.3.4.43 Date.prototype.toUTCString ()
This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent this time value in a convenient, human-readable form in UTC.

NOTE
The intent is to produce a String representation of a date that is more readable than the format specified in 20.3.16. It is not essential that the chosen format be unambiguous or easily machine parsable. If an implementation does not have a preferred human-readable format it is recommended to use the format defined in 20.3.16 but with a space rather than a "T" used to separate the date and time elements.

20.3.44 Date.prototype.valueOf()

The following steps are performed:

1. Return ? thisTimeValue(this value).

20.3.45 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 the String value "string" or the String value "default", then
   a. Let tryFirst be "string".
4. Else if hint is the String value "number", then
   a. Let tryFirst be "number".
5. Else, throw a TypeError exception.

The value of the name property of this function is "[Symbol.toPrimitive]".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

20.3.5 Properties of Date Instances

Date instances are ordinary objects that inherit properties from the Date prototype object. Date instances also have a [[DateValue]] internal slot. The [[DateValue]] internal slot is the time value represented by this Date object.
21 Text Processing

21.1 String Objects

21.1.1 The String Constructor

The String constructor is the %String% intrinsic object and the initial value of the String property of the global object. When called as a constructor it creates and initializes a new String object. When String is called as a function rather than as a constructor, it performs a type conversion.

The String 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.

21.1.1.1 String (value)

When String is called with argument value, the following steps are taken:

1. If no arguments were passed to this function invocation, let s be "".
2. Else,
   a. If NewTarget is undefined and Type(value) is Symbol, return SymbolDescriptiveString(value).
   b. Let s be ? ToString(value).
3. If NewTarget is undefined, return s.
4. Return ? StringCreate(s, ? GetPrototypeFromConstructor(NewTarget, "%StringPrototype%"))

21.1.2 Properties of the String Constructor

The value of the [[Prototype]] internal slot of the String constructor is the intrinsic object %FunctionPrototype%.

The String constructor has the following properties:

21.1.2.1 String.fromCharCode ( ...codeUnits )

The String.fromCharCode function may be called with any number of arguments which form the rest parameter codeUnits. The following steps are taken:

1. Let codeUnits be a List containing the arguments passed to this function.
2. Let length be the number of elements in codeUnits.
3. Let `elements` be a new empty `List`.
4. Let `nextIndex` be 0.
5. Repeat, while `nextIndex < length`
   a. Let `next` be `codeUnits[nextIndex]`.
   b. Let `nextCU` be `? ToUint16(next)`.
   c. Append `nextCU` to the end of `elements`.
   d. Let `nextIndex` be `nextIndex + 1`.
6. Return the String value whose elements are, in order, the elements in the `List elements`. If `length` is 0, the empty string is returned.

The `length` property of the `fromCharCode` function is 1.

### 21.1.2.2 String.fromCharCode ( ...`codePoints` )

The `String.fromCharCode` function may be called with any number of arguments which form the rest parameter `codePoints`. The following steps are taken:

1. Let `codePoints` be a `List` containing the arguments passed to this function.
2. Let `length` be the number of elements in `codePoints`.
3. Let `elements` be a new empty `List`.
4. Let `nextIndex` be 0.
5. Repeat, while `nextIndex < length`
   a. Let `next` be `codePoints[nextIndex]`.
   b. Let `nextCP` be `? ToNumber(next)`.
   c. If `SameValue(nextCP, ToInteger(nextCP))` is `false`, throw a `RangeError` exception.
   d. If `nextCP < 0` or `nextCP > 0x10FFFF`, throw a `RangeError` exception.
   e. Append the elements of the `UTF16Encoding` of `nextCP` to the end of `elements`.
   f. Let `nextIndex` be `nextIndex + 1`.
6. Return the String value whose elements are, in order, the elements in the `List elements`. If `length` is 0, the empty string is returned.

The `length` property of the `fromCharCode` function is 1.

### 21.1.2.3 String.prototype

The initial value of `String.prototype` is the intrinsic object `%StringPrototype%`.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 21.1.2.4 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 `substitutions` be a `List` consisting of all of the arguments passed to this function, starting with the second argument. If fewer than two arguments were passed, the `List` is empty.
2. Let `numberOfSubstitutions` be the number of elements in `substitutions`.
3. Let `cooked` be `ToObject(template)`.
4. Let `raw` be `ToObject(? Get(cooked, "raw"))`.
5. Let `literalSegments` be `ToLength(? Get(raw, "length"))`.
6. If `literalSegments` ≤ 0, return the empty string.
7. Let `stringElements` be a new empty `List`.
8. Let `nextIndex` be 0.
9. Repeat,
   a. Let `nextKey` be `ToString(nextIndex)`.
   b. Let `nextSeg` be `ToObject(? Get(raw, nextKey))`.
   c. Append in order 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, in order, 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 in order the code unit elements of `nextSub` to the end of `stringElements`.
   i. Let `nextIndex` be `nextIndex + 1`.

**NOTE** String.raw is intended for use as a tag function of a Tagged Template (12.3.7). When called as such, the first argument will be a well formed template object and the rest parameter will contain the substitution values.

### 21.1.3 Properties of the String Prototype Object

The String prototype object is the intrinsic object `%StringPrototype%`. The String prototype object is a String exotic object and has the internal methods specified for such objects. It has a `[[StringData]]` internal slot with the value `""`. It has a `length` property whose initial value is 0 and whose attributes are `{ [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

The value of the `[[Prototype]]` internal slot of the String prototype object is the intrinsic object `%ObjectPrototype%`.

Unless explicitly stated otherwise, the methods of the String prototype object defined below are not generic and the `this` value passed to them must be either a String value or an object that has a
[[StringData]] internal slot that has been initialized to a String value.

The abstract operation `thisStringValue(value)` performs the following steps:

1. If `Type(value)` is String, return `value`.
2. If `Type(value)` is Object and `value` has a `[[StringData]]` internal slot, then
   a. Assert: `value.[[StringData]]` is a String value.
   b. Return `value.[[StringData]]`.
3. Throw a `TypeError` exception.

### 21.1.3.1 String.prototype.charAt (pos)

**NOTE 1** Returns a single element String containing the code unit at index `pos` in 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 a value of Number type that is an integer, then the result of `x.charAt(pos)` is equal to the result of `x.substring(pos, pos+1).

When the `charAt` method is called with one argument `pos`, the following steps are taken:

1. Let `O` be ? `RequireObjectCoercible(this value)`.
2. Let `S` be ? `ToString(O)`.
3. Let `position` be ? `ToInteger(pos)`.
4. Let `size` be the number of elements in `S`.
5. If `position < 0` or `position ≥ size`, return the empty String.
6. Return a String 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.

### 21.1.3.2 String.prototype.charCodeAt (pos)

**NOTE 1** Returns a Number (a nonnegative integer less than $2^{16}$) that is the code unit value of the string element at index `pos` in 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 ? `ToInteger(pos)`.
4. Let `size` be the number of elements in `S`.
5. If \( \text{position} < 0 \) or \( \text{position} \geq \text{size} \), return \text{NaN}.
6. Return a value of Number type, whose value is the code unit value of the element at index \( \text{position} \) in the String \( S \).

**NOTE 2** The \texttt{charCodeAt} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

### 21.1.3.3 \texttt{String.prototype.codePointAt ( \textit{pos} )}

**NOTE 1** Returns a nonnegative integer Number less than 0x110000 that is the code point value of the UTF-16 encoded code point (6.1.4) starting at the string element at index \( \text{pos} \) in 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 \( \text{pos} \), the result is the code unit at \( \text{pos} \).

When the \texttt{codePointAt} method is called with one argument \( \text{pos} \), the following steps are taken:

1. Let \( O \) be \texttt{RequireObjectCoercible(this value)}.
2. Let \( S \) be \texttt{ToString(O)}.
3. Let \( \text{position} \) be \texttt{ToInteger(pos)}.
4. Let \( \text{size} \) be the number of elements in \( S \).
5. If \( \text{position} < 0 \) or \( \text{position} \geq \text{size} \), return \texttt{undefined}.
6. Let \( \text{first} \) be the code unit value of the element at index \( \text{position} \) in the String \( S \).
7. If \( \text{first} < 0x{D800} \) or \( \text{first} > 0xDBFF \) or \( \text{position}+1 = \text{size} \), return \( \text{first} \).
8. Let \( \text{second} \) be the code unit value of the element at index \( \text{position}+1 \) in the String \( S \).
9. If \( \text{second} < 0xDC00 \) or \( \text{second} > 0xDFFF \), return \( \text{first} \).
10. Return \texttt{UTF16Decode(first, second)}.

**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.

### 21.1.3.4 \texttt{String.prototype.concat ( ...\textit{args} )}

**NOTE 1** When the \texttt{concat} method is called it returns a String consisting of the code units of the \texttt{this} object (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 $args$ be a List whose elements are the arguments passed to this function.
4. Let $R$ be $S$.
5. Repeat, while $args$ is not empty
   a. Remove the first element from $args$ and let $next$ be the value of that element.
   b. Let $nextString$ be $? \text{ToString}(next)$.
   c. Set $R$ to the String value consisting of the code units of the previous value of $R$ followed by the code units of $nextString$.
6. Return $R$.

The `length` property of the `concat` method is 1.

**Note 2** The `concat` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

### 21.1.3.5 `String.prototype.constructor`

The initial value of `String.prototype.constructor` is the intrinsic object `%String%`.

### 21.1.3.6 `String.prototype.endsWith (searchString [, endPosition ] )`  

The following steps are taken:

1. Let $O$ be $? \text{RequireObjectCoercible}(this \text{ value})$.
2. Let $S$ be $? \text{ToString}(O)$.
3. Let $isRegExp$ be $? \text{IsRegExp}(searchString)$.
4. If $isRegExp$ is `true`, throw a `TypeError` exception.
5. Let $searchStr$ be $? \text{ToString}(searchString)$.
6. Let $len$ be the number of elements in $S$.
7. If $endPosition$ is `undefined`, let $pos$ be $len$, else let $pos$ be $? \text{ToInteger}(endPosition)$.
8. Let $end$ be $\text{min}(\text{max}(pos, 0), len)$.
9. Let $searchLength$ be the number of elements in $searchStr$.
10. Let $start$ be $\text{end} - searchLength$.
11. If $start$ is less than 0, return `false`.
12. If the sequence of elements of $S$ starting at $start$ of length $searchLength$ is the same as the full element sequence of $searchStr$, return `true`.
13. Otherwise, return `false`.

**Note 1** Returns `true` if the sequence of elements of $searchString$ converted to a String is the same as the corresponding elements of this object (converted to a String) starting at `endPosition` - length(this). Otherwise returns `false`.

**Note 2** Throwing an exception if the first argument is a RegExp is specified in order to allow
null
3. Let `searchStr` be `ToString(searchString)`.
4. Let `pos` be `ToInteger(position)`. (If `position` is `undefined`, this step produces the value 0.)
5. Let `len` be the number of elements in `S`.
6. Let `start` be `min(max(pos, 0), len)`.
7. Let `searchLen` be the number of elements in `searchStr`.
8. Return the smallest possible integer `k` not smaller than `start` such that `k+searchLen` is not greater than `len`, and for all nonnegative integers `j` less than `searchLen`, the code unit at index `k+j` of `S` is the same as the code unit at index `j` of `searchStr`; but if there is no such integer `k`, return the value -1.

| NOTE 2 | The `indexOf` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method. |

### 21.1.3.9 String.prototype.lastIndexOf ( `searchString` , `position` )

| NOTE 1 | If `searchString` appears as a substring of the result of converting this object to a String at one or more indices that are smaller than or equal to `position`, then the greatest such index is returned; otherwise, -1 is returned. If `position` is `undefined`, the length of the String value is assumed, so as to search all of the String. |

The `lastIndexOf` method takes two arguments, `searchString` and `position`, and performs the following steps:

1. Let `O` be `RequireObjectCoercible(this value)`.
2. Let `S` be `ToString(O)`.
3. Let `searchStr` be `ToString(searchString)`.
4. Let `numPos` be `ToNumber(position)`. (If `position` is `undefined`, this step produces the value `NaN`.)
5. If `numPos` is `NaN`, let `pos` be `+∞`; otherwise, let `pos` be `ToInteger(numPos)`.
6. Let `len` be the number of elements in `S`.
7. Let `start` be `min(max(pos, 0), len)`.
8. Let `searchLen` be the number of elements in `searchStr`.
9. Return the largest possible nonnegative integer `k` not larger than `start` such that `k+searchLen` is not greater than `len`, and for all nonnegative integers `j` less than `searchLen`, the code unit at index `k+j` of `S` is the same as the code unit at index `j` of `searchStr`; but if there is no such integer `k`, return the value -1.

| 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. |

### 21.1.3.10 String.prototype.localeCompare ( `that` [, `reserved1` [, `reserved2` ]] )
An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the `localeCompare` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `localeCompare` method is used.

When the `localeCompare` method is called with argument `that`, it returns a Number other than `NaN` that represents the result of a locale-sensitive String comparison of the `this` value (converted to a String) with `that` (converted to a String). The two Strings are `S` and `That`. The two Strings are compared in an implementation-defined fashion. The result is intended to order String values in the sort order specified by a host default locale, and will be negative, zero, or positive, depending on whether `S` comes before `That` in the sort order, the Strings are equal, or `S` comes after `That` in the sort order, respectively.

Before performing the comparisons, the following steps are performed to prepare the Strings:

1. Let `O` be ? `RequireObjectCoercible(this value)`.
2. Let `S` be ? `ToString(O)`.
3. Let `That` be ? `ToString(that)`.

The meaning of the optional second and third parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not assign any other interpretation to those parameter positions.

The `localeCompare` method, if considered as a function of two arguments `this` and `that`, is a consistent comparison function (as defined in 22.1.3.25) 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 (http://www.unicode.org/reports/tr15/) and Unicode Technical Note #5, Canonical Equivalence in Applications (http://www.unicode.org/notes/tn5/). Also see Unicode
NOTE 3 The `localeCompare` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.11 `String.prototype.match (regexp)`

When the `match` method is called with argument `regexp`, the following steps are taken:

1. Let `O` be `RequireObjectCoercible(this value)`.
2. 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.

21.1.3.12 `String.prototype.normalize ([form])`

When the `normalize` method is called with one argument `form`, the following steps are taken:

1. Let `O` be `RequireObjectCoercible(this value)`.
2. Let `S` be `ToString(O)`.
3. If `form` is not provided or `form` is `undefined`, let `form` be "NFC".
4. 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 `http://www.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.

21.1.3.13 `String.prototype.padEnd(maxLength [, fillString])`

When the `padEnd` method is called, the following steps are taken:
1. Let \( O \) be \( \text{RequireObjectCoercible}(\text{this} \ value) \).
2. Let \( S \) be \( \text{ToString}(O) \).
3. Let \( \text{intMaxLength} \) be \( \text{ToLength}(\text{maxLength}) \).
4. Let \( \text{stringLength} \) be the number of elements in \( S \).
5. If \( \text{intMaxLength} \) is not greater than \( \text{stringLength} \), return \( S \).
6. If \( \text{fillString} \) is \text{undefined}, let \( \text{filler} \) be a String consisting solely of the code unit \( 0x0020 \) (SPACE).
7. Else, let \( \text{filler} \) be \( \text{ToString}(\text{fillString}) \).
8. If \( \text{filler} \) is the empty String, return \( S \).
9. Let \( \text{fillLen} \) be \( \text{intMaxLength} - \text{stringLength} \).
10. Let \( \text{truncatedStringFiller} \) be a new String value consisting of repeated concatenations of \( \text{filler} \) truncated to length \( \text{fillLen} \).
11. Return a new String value computed by the concatenation of \( S \) and \( \text{truncatedStringFiller} \).

**NOTE 1** The first argument \( \text{maxLength} \) will be clamped such that it can be no smaller than the length of the \( \text{this} \) value.

**NOTE 2** The optional second argument \( \text{fillString} \) defaults to " " (a String consisting of \( 0x0020 \) SPACE).

### 21.1.3.14 String.prototype.padStart( maxLength [, fillString ] )

When the \( \text{padStart} \) method is called, the following steps are taken:

1. Let \( O \) be \( \text{RequireObjectCoercible}(\text{this} \ value) \).
2. Let \( S \) be \( \text{ToString}(O) \).
3. Let \( \text{intMaxLength} \) be \( \text{ToLength}(\text{maxLength}) \).
4. Let \( \text{stringLength} \) be the number of elements in \( S \).
5. If \( \text{intMaxLength} \) is not greater than \( \text{stringLength} \), return \( S \).
6. If \( \text{fillString} \) is \text{undefined}, let \( \text{filler} \) be a String consisting solely of the code unit \( 0x0020 \) (SPACE).
7. Else, let \( \text{filler} \) be \( \text{ToString}(\text{fillString}) \).
8. If \( \text{filler} \) is the empty String, return \( S \).
9. Let \( \text{fillLen} \) be \( \text{intMaxLength} - \text{stringLength} \).
10. Let \( \text{truncatedStringFiller} \) be a new String value consisting of repeated concatenations of \( \text{filler} \) truncated to length \( \text{fillLen} \).
11. Return a new String value computed by the concatenation of \( \text{truncatedStringFiller} \) and \( S \).

**NOTE 1** The first argument \( \text{maxLength} \) will be clamped such that it can be no smaller than the length of the \( \text{this} \) value.

**NOTE 2** The optional second argument \( \text{fillString} \) defaults to " " (a String consisting of \( 0x0020 \) SPACE).
21.1.3.15 String.prototype.repeat (\textit{count})

The following steps are taken:

1. Let $O$ be \texttt{RequireObjectCoercible(this value)}.
2. Let $S$ be \texttt{ToString($O$)}.
3. Let $n$ be \texttt{ToInteger(count)}.
4. If $n < 0$, throw a \texttt{RangeError} exception.
5. If $n$ is $+\infty$, throw a \texttt{RangeError} exception.
6. Let $T$ be a String value that is made from $n$ copies of $S$ appended together. If $n$ is 0, $T$ is the empty String.
7. Return $T$.

\textbf{NOTE 1} This method creates a String consisting of the code units of the \texttt{this} object (converted to String) repeated \textit{count} times.

\textbf{NOTE 2} The \texttt{repeat} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.16 String.prototype.replace (\textit{searchValue}, \textit{replaceValue})

When the \texttt{replace} method is called with arguments \textit{searchValue} and \textit{replaceValue}, the following steps are taken:

1. Let $O$ be \texttt{RequireObjectCoercible(this value)}.
2. If \textit{searchValue} is neither \texttt{undefined} nor \texttt{null}, then
   a. Let \texttt{replacer} be \texttt{GetMethod(searchValue,@@replace)}.
   b. If \texttt{replacer} is not \texttt{undefined}, then
      i. Return ? \texttt{Call(replacer, searchValue, « $O$, replaceValue »)}.
3. Let \textit{string} be \texttt{ToString($O$)}.
4. Let \textit{searchString} be \texttt{ToString(searchValue)}.
5. Let \texttt{functionalReplace} be \texttt{IsCallable(replaceValue)}.
6. If \texttt{functionalReplace} is \texttt{false}, then
   a. Let \texttt{replaceValue} be \texttt{ToString(replaceValue)}.
7. Search \textit{string} for the first occurrence of \textit{searchString} and let \textit{pos} be the index within \textit{string} of the first code unit of the matched substring and let \textit{matched} be \textit{searchString}. If no occurrences of \textit{searchString} were found, return \textit{string}.
8. If \texttt{functionalReplace} is \texttt{true}, then
   a. Let \texttt{replValue} be ? \texttt{Call(replaceValue, undefined, « matched, pos, string »)}.
   b. Let \texttt{replStr} be \texttt{ToString(replValue)}.
9. Else,
   a. Let \texttt{captures} be a new empty \texttt{List}.
b. Let `replStr` be `GetSubstitution(matched, string, pos, captures, replaceValue).

10. Let `tailPos` be `pos + the number of code units in matched`.

11. Let `newString` be the String formed by concatenating the first `pos` code units of `string`, `replStr`, and the trailing substring of `string` starting at index `tailPos`. If `pos` is 0, the first element of the concatenation will be the empty String.

12. Return `newString`.

**NOTE** The replace function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.16.1 Runtime Semantics: `GetSubstitution(matched, str, position, captures, replacement)`

The abstract operation GetSubstitution performs the following steps:

1. Assert: Type(`matched`) is String.
2. Let `matchLength` be the number of code units in `matched`.
3. Assert: Type(`str`) is String.
4. Let `stringLength` be the number of code units in `str`.
5. Assert: `position` is a nonnegative integer.
7. Assert: `captures` is a possibly empty List of Strings.
8. Assert: Type(`replacement`) is String.
9. Let `tailPos` be `position + matchLength`.
10. Let `m` be the number of elements in `captures`.

11. Let `result` be a String value derived from `replacement` by copying code unit elements from `replacement` to `result` while performing replacements as specified in Table 46. These replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements.

12. Return `result`.

**Table 46: Replacement Text Symbol Substitutions**

<table>
<thead>
<tr>
<th>Code units</th>
<th>Unicode Characters</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0024, 0x0024</td>
<td>$$</td>
<td>$</td>
</tr>
<tr>
<td>0x0024, 0x0026</td>
<td>$&amp;</td>
<td>matched</td>
</tr>
<tr>
<td>0x0024, 0x0060</td>
<td>$`</td>
<td>If <code>position</code> is 0, the replacement is the empty String. Otherwise the replacement is the substring of <code>str</code> that starts at index 0 and whose last code unit is at index <code>position - 1</code>.</td>
</tr>
</tbody>
</table>
If `tailPos ≥ stringLength`, the replacement is the empty String. Otherwise the replacement is the substring of `str` that starts at index `tailPos` and continues to the end of `str`.

The `n`\textsuperscript{th} element of `captures`, where `n` is a single digit in the range 1 to 9. If `n ≤ m` and the `n`\textsuperscript{th} element of `captures` is `undefined`, use the empty String instead. If `n > m`, the result is implementation-defined.

The `nn`\textsuperscript{th} element of `captures`, where `nn` is a two-digit decimal number in the range 01 to 99. If `nn ≤ m` and the `nn`\textsuperscript{th} element of `captures` is `undefined`, use the empty String instead. If `nn` is 00 or `nn > m`, the result is implementation-defined.

$\in$ in any context that does not match any of the above.

### 21.1.3.17 String.prototype.search (regexp)

When the search method is called with argument `regexp`, the following steps are taken:

1. Let `O` be `? RequireObjectCoercible(this value)`. 
2. If `regexp` is neither `undefined` nor `null`, then 
   a. Let `searcher` be `? GetMethod(regexp, @@search)`. 
   b. If `searcher` is not `undefined`, then 
      i. Return `? Call(searcher, regexp, « O »)`.
3. Let `string` be `? ToString(O)`.
4. Let `rx` be `? RegExpCreate(regexp, undefined)`.

**NOTE** The `search` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.18 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 ? `ToObjectCoercible(this value).`
2. Let `S` be ? `ToString(O).`
3. Let `len` be the number of elements in `S`.
4. Let `intStart` be ? `ToInteger(start).`
5. If `end` is undefined, let `intEnd` be `len`; else let `intEnd` be ? `ToInteger(end).`
6. If `intStart` < 0, let `from` be `max(len + intStart, 0)`; otherwise let `from` be `min(intStart, len).`
7. If `intEnd` < 0, let `to` be `max(len + intEnd, 0)`; otherwise let `to` be `min(intEnd, len).`
8. Let `span` be `max(to - from, 0).`
9. Return a String value containing `span` consecutive elements from `S` beginning with the element at index `from`.

### NOTE

The `slice` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

### 21.1.3.19 String.prototype.split ( `separator`, `limit` )

Returns an Array object into which substrings of the result of converting this object to a String have been stored. The substrings are determined by searching from left to right for occurrences of `separator`; these occurrences are not part of any substring in the returned array, but serve to divide up the String value. The value of `separator` may be a String of any length or it may be 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 ? `ToObjectCoercible(this value).`
2. If `separator` is neither undefined nor null, then
      b. If `splitter` is not undefined, then
         i. Return ? `Call(splitter, separator, « O, limit »).`
3. Let `S` be ? `ToString(O).`
4. Let `A` be ! `ArrayCreate(0).`
5. Let `lengthA` be 0.
6. If `limit` is undefined, let `lim` be `2^{32}-1`; else let `lim` be ? `ToUint32(limit).`
7. Let `s` be the number of elements in `S`.
8. Let `p` be 0.
9. Let \( R \) be ? \( \text{ToString}(\text{separator}) \).

10. If \( \text{lim} = 0 \), return \( A \).

11. If \( \text{separator} \) is \textbf{undefined}, then
   a. Perform \( \text{!CreateDataProperty}(A, "\empty", S) \).
   b. Return \( A \).

12. If \( s = 0 \), then
   a. Let \( z \) be \( \text{SplitMatch}(S, 0, R) \).
   b. If \( z \) is not \textbf{false}, return \( A \).
   c. Perform \( \text{!CreateDataProperty}(A, "\empty", S) \).
   d. Return \( A \).

13. Let \( q \) be \( p \).

14. Repeat, while \( q \neq s \)
   a. Let \( e \) be \( \text{SplitMatch}(S, q, R) \).
   b. If \( e \) is \textbf{false}, let \( q \) be \( q+1 \).
   c. Else \( e \) is an integer index \( \leq s \),
      i. If \( e = p \), let \( q \) be \( q+1 \).
      ii. Else \( e \neq p \),
          1. Let \( T \) be a String value equal to the substring of \( S \) consisting of the code units at indices \( p \) (inclusive) through \( q \) (exclusive).
          2. Perform \( \text{!CreateDataProperty}(A, \text{!ToString}(\text{lengthA}), T) \).
          3. Increment \( \text{lengthA} \) by 1.
          4. If \( \text{lengthA} = \text{lim} \), return \( A \).
          5. Let \( p \) be \( e \).
          6. Let \( q \) be \( p \).

15. Let \( T \) be a String value equal to the substring of \( S \) consisting of the code units at indices \( p \) (inclusive) through \( s \) (exclusive).

16. Perform \( \text{!CreateDataProperty}(A, \text{!ToString}(\text{lengthA}), T) \).

17. Return \( A \).

\textbf{NOTE 1}  

The value of \( \text{separator} \) may be an empty String. In this case, \( \text{separator} \) does not match the empty substring at the beginning or end of the input String, nor does it match the empty substring at the end of the previous separator match. If \( \text{separator} \) is the empty String, the String is split up into individual code unit elements; the length of the result array equals the length of the String, and each substring contains one code unit.

If the \textbf{this} object is (or converts to) the empty String, the result depends on whether \( \text{separator} \) can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

If \( \text{separator} \) is \textbf{undefined}, then the result array contains just one String, which is the \textbf{this} value (converted to a String). If \( \text{limit} \) is not \textbf{undefined}, then the output array is truncated...
so that it contains no more than \textit{limit} elements.

\textbf{NOTE 2} \hspace{1em} The \texttt{split} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.19.1 Runtime Semantics: SplitMatch ( \( S, q, R \) )

The abstract operation \texttt{SplitMatch} takes three parameters, a String \( S \), an integer \( q \), and a String \( R \), and performs the following steps in order to return either \texttt{false} or the end index of a match:

1. Assert: \texttt{Type}(\( R \)) is \texttt{String}.
2. Let \( r \) be the number of code units in \( R \).
3. Let \( s \) be the number of code units in \( S \).
4. If \( q+r > s \), return \texttt{false}.
5. If there exists an integer \( i \) between 0 (inclusive) and \( r \) (exclusive) such that the code unit at index \( q+i \) of \( S \) is different from the code unit at index \( i \) of \( R \), return \texttt{false}.
6. Return \( q+r \).

### 21.1.3.20 String.prototype.startsWith ( \texttt{searchString} [ , \texttt{position} ] )

The following steps are taken:

1. Let \( O \) be \texttt{RequireObjectCoercible}(\texttt{this} value).
2. Let \( S \) be \texttt{ToString}(\( O \)).
3. Let \( isRegExp \) be \texttt{IsRegExp}(\texttt{searchString}).
4. If \( isRegExp \) is \texttt{true}, throw a \texttt{TypeError} exception.
5. Let \( searchString \) be \texttt{ToString}(\texttt{searchString}).
6. Let \( pos \) be \texttt{ToInteger}(\texttt{position}). (If \texttt{position} is \texttt{undefined}, this step produces the value 0.)
7. Let \( len \) be the number of elements in \( S \).
8. Let \( start \) be \texttt{min}(\texttt{max}(\( pos \), 0), \( len \)).
9. Let \( searchLength \) be the number of elements in \( searchString \).
10. If \( searchLength+start \) is greater than \( len \), return \texttt{false}.
11. If the sequence of elements of \( S \) starting at \( start \) of length \( searchLength \) is the same as the full element sequence of \( searchString \), return \texttt{true}.
12. Otherwise, return \texttt{false}.

\textbf{NOTE 1} \hspace{1em} This method returns \texttt{true} if the sequence of elements of \texttt{searchString} converted to a String is the same as the corresponding elements of this object (converted to a String) starting at index \texttt{position}. Otherwise returns \texttt{false}.

\textbf{NOTE 2} \hspace{1em} 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.
The `startsWith` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.21 String.prototype.substring ( `start, end` )

The `substring` method takes two arguments, `start` and `end`, and returns a substring of the result of converting this object to a String, starting from index `start` and running to, but not including, index `end` of the String (or through the end of the String if `end` is `undefined`). The result is a String value, not a String object.

If either argument is `NaN` or negative, it is replaced with zero; if either argument is larger than the length of the String, it is replaced with the length of the String.

If `start` is larger than `end`, they are swapped.

The following steps are taken:

1. Let `O` be ? `RequireObjectCoercible(this value)`.
2. Let `S` be ? `ToString(O)`.
3. Let `len` be the number of elements in `S`.
4. Let `intStart` be ? `ToInteger(start)`.
5. If `end` is `undefined`, let `intEnd` be `len`; else let `intEnd` be ? `ToInteger(end)`.
6. Let `finalStart` be `min(max(intStart, 0), len)`.
7. Let `finalEnd` be `min(max(intEnd, 0), len)`.
8. Let `from` be `min(finalStart, finalEnd)`.
9. Let `to` be `max(finalStart, finalEnd)`.
10. Return a String whose length is `to - from`, containing code units from `S`, namely the code units with indices `from` through `to - 1`, in ascending order.

The `substring` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.22 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.
```

### 21.1.3.23 `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.
```

### 21.1.3.24 `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 \( cpList \) be a List containing in order the code points as defined in 6.1.4 of \( S \), starting at the first element of \( S \).

4. For each code point \( c \) in \( cpList \), if the Unicode Character Database provides a language insensitive lower case equivalent of \( c \), then replace \( c \) in \( cpList \) with that equivalent code point(s).

5. Let \( cuList \) be a new empty List.

6. For each code point \( c \) in \( cpList \), in order, append to \( cuList \) the elements of the UTF16Encoding of \( c \).

7. Let \( L \) be a String whose elements are, in order, the elements of \( cuList \).

8. Return \( L \).

The result must be derived according to the locale-insensitive case mappings in the Unicode Character Database (this explicitly includes not only the UnicodeData.txt file, but also all locale-insensitive mappings in the SpecialCasings.txt file that accompanies it).

**NOTE 1**

The case mapping of some code points may produce multiple code points. In this case the result String may not be the same length as the source String. Because both `toUpperCase` and `toLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, `s.toUpperCase().toLowerCase()` is not necessarily equal to `s.toLowerCase()`.

**NOTE 2**

The `toLowerCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.25 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.

### 21.1.3.26 String.prototype.toUpperCase ()

This function interprets a String value as a sequence of UTF-16 encoded code points, as described in 6.1.4.

This function behaves in exactly the same way as `String.prototype.toLowerCase`, except that code points are mapped to their `uppercase` equivalents as specified in the Unicode Character Database.

**NOTE**

The `toUpperCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.
### 21.1.3.27 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 \( O \) be \? RequireObjectCoercible(\texttt{this} value).
2. Let \( S \) be \? ToString(\( O \)).
3. Let \( T \) be a String value that is a copy of \( S \) with both leading and trailing white space removed. The definition of white space is the union of \textit{WhiteSpace} and \textit{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.
4. Return \( T \).

**NOTE** The \texttt{trim} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.28 String.prototype.valueOf ()

When the \texttt{valueOf} method is called, the following steps are taken:

1. Return \? thisStringValue(\texttt{this} value).

### 21.1.3.29 String.prototype[ @@iterator ] ()

When the \texttt{@@iterator} method is called it returns an Iterator object (25.1.1.2) that iterates over the code points of a String value, returning each code point as a String value. The following steps are taken:

1. Let \( O \) be \? RequireObjectCoercible(\texttt{this} value).
2. Let \( S \) be \? ToString(\( O \)).
3. Return CreateStringIterator(\( S \)).

The value of the \texttt{name} property of this function is "\texttt{[Symbol.iterator]}".

### 21.1.4 Properties of String Instances

String instances are String exotic objects and have the internal methods specified for such objects. String instances inherit properties from the String prototype object. String instances also have a [[StringData]] internal slot.

String instances have a \texttt{length} property, and a set of enumerable properties with integer indexed names.

#### 21.1.4.1 length
The number of elements in the String value represented by this String object.

Once a String object is initialized, this property is unchanging. It has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 21.1.5 String Iterator Objects

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.

#### 21.1.5.1 CreateStringIterator (\texttt{string})

Several methods of String objects return Iterator objects. The abstract operation CreateStringIterator with argument \texttt{string} is used to create such iterator objects. It performs the following steps:

1. Assert: \texttt{Type(string)} is String.
2. Let \texttt{iterator} be ObjectCreate(%StringIteratorPrototype%, « [[IteratedString]], [[StringIteratorNextIndex]] »).
3. Set \texttt{iterator}[[IteratedString]] to \texttt{string}.
4. Set \texttt{iterator}[[StringIteratorNextIndex]] to 0.
5. Return \texttt{iterator}.

#### 21.1.5.2 The %StringIteratorPrototype% Object

All String Iterator Objects inherit properties from the %StringIteratorPrototype% intrinsic object. The %StringIteratorPrototype% object is an ordinary object and its [[Prototype]] internal slot is the %IteratorPrototype% intrinsic object. In addition, %StringIteratorPrototype% has the following properties:

##### 21.1.5.2.1 %StringIteratorPrototype%.\texttt{next} ( )

1. Let \texttt{O} be the \texttt{this} value.
2. If \texttt{Type(O)} is not Object, throw a \texttt{TypeError} exception.
3. If \texttt{O} does not have all of the internal slots of a String Iterator Instance (21.1.5.3), throw a \texttt{TypeError} exception.
4. Let \texttt{s} be \texttt{O}[[IteratedString]].
5. If \texttt{s} is \texttt{undefined}, return CreatelterResultObject(\texttt{undefined}, \texttt{true}).
6. Let \texttt{position} be \texttt{O}[[StringIteratorNextIndex]].
7. Let \texttt{len} be the number of elements in \texttt{s}.
8. If \texttt{position} \geq \texttt{len}, then
a. Set $O.[[\text{IteratedString}]]$ to undefined.
b. Return $\text{Create ITER Result Object(undefined, true)}$.

9. Let first be the code unit value at index $position$ in $s$.
10. If $first < 0x80$ or $first > 0xDBFF$ or $position + 1 = len$, let $resultString$ be the String consisting of the single code unit first.
11. Else,
   a. Let second be the code unit value at index $position + 1$ in the String $S$.
   b. If $second < 0xDC00$ or $second > 0xDFFF$, let $resultString$ be the String consisting of the single code unit first.
   c. Else, let $resultString$ be the String consisting of the code unit first followed by the code unit second.
12. Let $resultSize$ be the number of code units in $resultString$.
13. Set $O.[[\text{String Iterator Next Index}]]$ to $position + resultSize$.
14. Return $\text{Create ITER Result Object(resultString, false)}$.

21.1.5.2.2 %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 }.

21.1.5.3 Properties of String Iterator Instances

String Iterator instances are ordinary objects that inherit properties from the %StringIteratorPrototype% intrinsic object. String Iterator instances are initially created with the internal slots listed in Table 47.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Iterated String]]</td>
<td>The String value whose elements are being iterated.</td>
</tr>
<tr>
<td>[[String Iterator Next Index]]</td>
<td>The integer index of the next string index to be examined by this iteration.</td>
</tr>
</tbody>
</table>

21.2 RegExp (Regular Expression) Objects

A RegExp object contains a regular expression and the associated flags.

**NOTE** The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

21.2.1 Patterns
The `RegExp` constructor applies the following grammar to the input pattern `String`. An error occurs if the grammar cannot interpret the `String` as an expansion of `Pattern`.

**Syntax**

\[
\begin{align*}
Pattern &::= \text{Disjunction} \\
\text{Disjunction} &::= \text{Alternative} | \text{Disjunction} \\
\text{Alternative} &::= \text{empty} | \text{Alternative} \text{Term} \\
\text{Term} &::= \text{Assertion} | \text{Atom} | \text{Atom} \text{Quantifier} \\
\text{Assertion} &::= ^ | $ | \backslash \text{b} | \backslash \text{B} | (? = \text{Disjunction}) | (? ! \text{Disjunction}) \\
\text{Quantifier} &::= \text{QuantifierPrefix} \\
\text{QuantifierPrefix} &::= \* | + | ? \\
\end{align*}
\]

\[
\begin{align*}
\text{DecimalDigits} &::= \{ \text{DecimalDigits} \} \\
\text{DecimalDigits} &::= \{ \text{DecimalDigits}, \} \\
\text{DecimalDigits} &::= \{ \text{DecimalDigits}, \text{DecimalDigits} \}
\end{align*}
\]

\[
\text{Atom}::= \text{Syntax}
\]
PatternCharacter
    .
    \ AtomEscape[?U]
CharacterClass[?U]
    ( Disjunction[?U] )
    ( ? : Disjunction[?U] )

SyntaxCharacter :: one of
    ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
    SourceCharacter but not SyntaxCharacter

AtomEscape[?U] ::
    DecimalEscape
    CharacterClassEscape
    CharacterEscape[?U]

CharacterEscape[?U] ::
    ControlEscape
    c ControlLetter
    Θ [lookahead ∈ DecimalDigit]
    HexEscapeSequence
    RegExpUnicodeEscapeSequence[?U]
    IdentityEscape[?U]

ControlEscape :: one of
    fnrtv

ControlLetter :: one of
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

RegExpUnicodeEscapeSequence[?U] ::
    [+U] u LeadSurrogate \u TrailSurrogate
    [+U] u LeadSurrogate
    [+U] u TrailSurrogate
    [+U] u NonSurrogate
    [~U] u Hex4Digits
    [+U] u{ HexDigits }
Each TrailSurrogate for which the choice of associated LeadSurrogate is ambiguous shall be associated with the nearest possible LeadSurrogate that would otherwise have no corresponding TrailSurrogate.

LeadSurrogate ::
    Hex4Digits but only if the SV of Hex4Digits is in the inclusive range 0xD800 to 0xDBFF

TrailSurrogate ::
    Hex4Digits but only if the SV of Hex4Digits is in the inclusive range 0xDC00 to 0xDFFF

NonSurrogate ::
    Hex4Digits but only if the SV of Hex4Digits is not in the inclusive range 0xD800 to 0xDFFF

IdentityEscape[U] ::
    [+U] SyntaxCharacter
    [+U] /
    [-U] SourceCharacter but not UnicodeIDContinue

DecimalEscape ::
    NonZeroDigit DecimalDigits_opt [lookahead $\notin$ DecimalDigit]

CharacterClassEscape :: one of
    d D s S w W

CharacterClass[U] ::
    [ [lookahead $\notin$ {^}] ClassRanges[U] ]
    [ ^ ClassRanges[U] ]

ClassRanges[U] ::
    [empty]
    NonemptyClassRanges[U]

NonemptyClassRanges[U] ::
    ClassAtom[U]
    ClassAtom[U] NonemptyClassRangesNoDash[U]
    ClassAtom[U] – ClassAtom[U] ClassRanges[U]

NonemptyClassRangesNoDash[U] ::
    ClassAtom[U]
    ClassAtomNoDash[U] NonemptyClassRangesNoDash[U]
    ClassAtomNoDash[U] – ClassAtom[U] ClassRanges[U]

ClassAtom[U] ::
    -
\textit{ClassAtomNoDash}\[?u\]

\textit{ClassAtomNoDash}[u] ::

\textit{SourceCharacter but not one of \ or ] or -}

\textit{\ ClassEscape}[?u]

\textit{ClassEscape[u]} ::

b

\textit{[+u] -}

\textit{CharacterClassEscape}

\textit{CharacterEscape}[?u]

\subsection*{21.2.1.1 Static Semantics: Early Errors}

\textit{RegExpUnicodeEscapeSequence :: u\{ HexDigits \}}

- It is a Syntax Error if the MV of \textit{HexDigits} > 0x10FFFF.

\subsection*{21.2.2 Pattern Semantics}

A regular expression pattern is converted into an internal procedure using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same. The internal procedure is used as the value of a RegExp object's \textit{[[RegExpMatcher]]} internal slot.

A \textit{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 \textit{Pattern} is defined as if the source code for the \textit{Pattern} was a \textit{List} of \textit{SourceCharacter} values where each \textit{SourceCharacter} corresponds to a Unicode code point. If a BMP pattern contains a non-BMP \textit{SourceCharacter} the entire pattern is encoded using UTF-16 and the individual code units of that encoding are used as the elements of the \textit{List}.

\textbf{NOTE} 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) \textit{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 `UTF16Decode` must be used in producing a `List` consisting of a single pattern character, the code point U+1D11E.

An implementation may not actually perform such translations to or from UTF-16, but the semantics of this specification requires that the result of pattern matching be as if such translations were performed.

### 21.2.2.1 Notation

The descriptions below use the following variables:

- **Input** is a `List` consisting of all of the characters, in order, of the String being matched by the regular expression pattern. Each character is either a code unit or a code point, depending upon the kind of pattern involved. The notation `Input[n]` means the \( n \)th character of `Input`, where \( n \) can range between 0 (inclusive) and `InputLength` (exclusive).
- **InputLength** is the number of characters in `Input`.
- **NcapturingParens** is the total number of left-capturing parentheses (i.e. the total number of `Atom :: ( Disjunction )` Parse Nodes) in the pattern. A left-capturing parenthesis is any ( pattern character that is matched by the ( terminal of the `Atom :: ( Disjunction )` production.
- **IgnoreCase** is `true` if the RegExp object’s `[[OriginalFlags]]` internal slot contains "i" and otherwise is false.
- **Multiline** is `true` if the RegExp object’s `[[OriginalFlags]]` internal slot contains "m" and otherwise is false.
- **Unicode** is `true` if the RegExp object’s `[[OriginalFlags]]` internal slot contains "u" and otherwise is false.

Furthermore, the descriptions below use the following internal data structures:

- A `CharSet` is a mathematical set of characters, either code units or code points depending up the state of the `Unicode` flag. “All characters” means either all code unit values or all code point values also depending upon the state if `Unicode`.
- A `State` is an ordered pair \((\text{endIndex}, \text{captures})\) where `endIndex` is an integer and `captures` is a `List` of `NcapturingParens` values. States are used to represent partial match states in the regular expression matching algorithms. The `endIndex` is one plus the index of the last input character matched so far by
the pattern, while \textit{captures} holds the results of capturing parentheses. The \textit{n}\textsuperscript{th} element of \textit{captures} is either a \texttt{List} that represents the value obtained by the \textit{n}\textsuperscript{th} set of capturing parentheses or \texttt{undefined} if the \textit{n}\textsuperscript{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 \textit{MatchResult} is either a State or the special token \texttt{failure} that indicates that the match failed.
- A \textit{Continuation} procedure is an internal closure (i.e. an internal procedure with some arguments already bound to values) that takes one State argument and returns a MatchResult result. If an internal closure references variables which are bound in the function that creates the closure, the closure uses the values that these variables had at the time the closure was created. The Continuation attempts to match the remaining portion (specified by the closure’s already-bound arguments) of the pattern against \textit{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 \textit{Matcher} procedure is an internal closure that takes two arguments — a State and a Continuation — and returns a MatchResult result. A Matcher attempts to match a middle subpattern (specified by the closure’s already-bound arguments) of the pattern against \textit{Input}, starting at the intermediate state given by its State argument. The Continuation argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new State, the Matcher then calls Continuation on that new State to test if the rest of the pattern can match as well. If it can, the Matcher returns the State returned by Continuation; if not, the Matcher may try different choices at its choice points, repeatedly calling Continuation until it either succeeds or all possibilities have been exhausted.
- An \textit{AssertionTester} procedure is an internal closure that takes a State argument and returns a Boolean result. The assertion tester tests a specific condition (specified by the closure’s already-bound arguments) against the current place in \textit{Input} and returns \texttt{true} if the condition matched or \texttt{false} if not.

21.2.2.2 Pattern

The production \textit{Pattern} :: \textit{Disjunction} evaluates as follows:

1. Evaluate \textit{Disjunction} to obtain a Matcher \textit{m}.
2. Return an internal closure that takes two arguments, a String \textit{str} and an integer \textit{index}, and performs the following steps:
   a. Assert: \textit{index} ≤ the number of elements in \textit{str}.
   b. If \textit{Unicode} is \texttt{true}, let \textit{Input} be a \texttt{List} consisting of the sequence of code points of \textit{str} interpreted as a UTF-16 encoded (6.1.4) Unicode string. Otherwise, let \textit{Input} be a \texttt{List} consisting of the sequence of code units that are the elements of \textit{str}. \textit{Input} will be used throughout the algorithms in 21.2.2. Each element of \textit{Input} is considered to be a character.
c. Let $InputLength$ be the number of characters contained in $Input$. This variable will be used throughout the algorithms in 21.2.2.

d. Let $listIndex$ be the index into $Input$ of the character that was obtained from element $index$ of $str$.

e. Let $c$ be a Continuation that always returns its State argument as a successful MatchResult.

f. Let $cap$ be a List of $NcapturingParens$ undefined values, indexed 1 through $NcapturingParens$.

g. Let $x$ be the State ($listIndex$, $cap$).

h. Call $m(x, c)$ and return its result.

NOTE A Pattern evaluates (“compiles”) to an internal procedure 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 21.2.2 are designed so that compiling a pattern may throw a $SyntaxError$ exception; on the other hand, once the pattern is successfully compiled, applying the resulting internal procedure to find a match in a String cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

21.2.2.3 Disjunction

The production $Disjunction :: Alternative$ evaluates as follows:

1. Evaluate $Alternative$ to obtain a Matcher $m$.
2. Return $m$.

The production $Disjunction :: Alternative | Disjunction$ evaluates as follows:

1. Evaluate $Alternative$ to obtain a Matcher $m1$.
2. Evaluate $Disjunction$ to obtain a Matcher $m2$.
3. Return an internal Matcher closure that takes two arguments, a State $x$ and a Continuation $c$, and performs the following steps when evaluated:
   a. Call $m1(x, c)$ and let $r$ be its result.
   b. If $r$ is not $failure$, return $r$.
   c. Call $m2(x, c)$ and return its result.

NOTE The $|$ regular expression operator separates two alternatives. The pattern first tries to match the left $Alternative$ (followed by the sequel of the regular expression); if it fails, it tries to match the right $Disjunction$ (followed by the sequel of the regular expression). If the left $Alternative$, the right $Disjunction$, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left $Alternative$. If choices in the left $Alternative$ are exhausted, the right $Disjunction$ is tried instead of the left
*Alternative*. Any capturing parentheses inside a portion of the pattern skipped by | produce **undefined** values instead of Strings. Thus, for example,

```
/a|ab/.exec("abc")
```

returns the result "a" and not "ab". Moreover,

```
/((a)|(ab))((c)|(bc))/ .exec("abc")
```

returns the array

```
["abc", "a", "a", undefined, "bc", undefined, "bc"]
```

and not

```
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

### 21.2.2.4 Alternative

The production *Alternative* :: [empty] evaluates as follows:

1. Return a Matcher that takes two arguments, a State \( x \) and a Continuation \( c \), and returns the result of calling \( c(x) \).

The production *Alternative* :: *Alternative* *Term* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher \( m1 \).
2. Evaluate *Term* to obtain a Matcher \( m2 \).
3. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following steps when evaluated:
   a. Let \( d \) be a Continuation that takes a State argument \( y \) and returns the result of calling \( m2(y, c) \).
   b. Call \( m1(x, d) \) and return its result.

**NOTE** Consecutive *Terms* try to simultaneously match consecutive portions of *Input*. If the left *Alternative*, the right *Term*, and the sequel of the regular expression all have choice points, all choices in the sequel are tried before moving on to the next choice in the right *Term*, and all choices in the right *Term* are tried before moving on to the next choice in the left *Alternative*.

### 21.2.2.5 Term

The production *Term* :: *Assertion* evaluates as follows:
1. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following steps when evaluated:
   a. Evaluate \( \text{Assertion} \) to obtain an \( \text{Assertion} \text{Tester} \) \( t \).
   b. Call \( t(x) \) and let \( r \) be the resulting Boolean value.
   c. If \( r \) is \text{false}, return failure.
   d. Call \( c(x) \) and return its result.

The production \( \text{Term} ::= \text{Atom} \) evaluates as follows:

1. Return the Matcher that is the result of evaluating \( \text{Atom} \).

The production \( \text{Term} ::= \text{Atom} \text{ Quantifier} \) evaluates as follows:

1. Evaluate \( \text{Atom} \) to obtain a Matcher \( m \).
2. Evaluate \( \text{Quantifier} \) to obtain the three results: an integer \( \text{min} \), an integer (or \( \infty \)) \( \text{max} \), and Boolean \( \text{greedy} \).
3. If \( \text{max} \) is finite and less than \( \text{min} \), throw a \text{SyntaxError} exception.
4. Let \( \text{parenIndex} \) be the number of left-capturing parentheses in the entire regular expression that occur to the left of this \( \text{Term} \). This is the total number of \( \text{Atom} ::= (\text{Disjunction}) \) Parse Nodes prior to or enclosing this \( \text{Term} \).
5. Let \( \text{parenCount} \) be the number of left-capturing parentheses in \( \text{Atom} \). This is the total number of \( \text{Atom} ::= (\text{Disjunction}) \) Parse Nodes enclosed by \( \text{Atom} \).
6. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following steps when evaluated:
   a. Call \( \text{RepeatMatcher}(m, \text{min}, \text{max}, \text{greedy}, x, c, \text{parenIndex}, \text{parenCount}) \) and return its result.

21.2.2.5.1 Runtime Semantics: \( \text{RepeatMatcher}(m, \text{min}, \text{max}, \text{greedy}, x, c, \text{parenIndex}, \text{parenCount}) \)

The abstract operation \( \text{RepeatMatcher} \) takes eight parameters, a Matcher \( m \), an integer \( \text{min} \), an integer (or \( \infty \)) \( \text{max} \), a Boolean \( \text{greedy} \), a State \( x \), a Continuation \( c \), an integer \( \text{parenIndex} \), and an integer \( \text{parenCount} \), and performs the following steps:

1. If \( \text{max} \) is zero, return \( c(x) \).
2. Let \( d \) be an internal Continuation closure that takes one State argument \( y \) and performs the following steps when evaluated:
   a. If \( \text{min} \) is zero and \( y \)'s endIndex is equal to \( x \)'s endIndex, return failure.
   b. If \( \text{min} \) is zero, let \( \text{min}2 \) be zero; otherwise let \( \text{min}2 \) be \( \text{min}-1 \).
   c. If \( \text{max} \) is \( \infty \), let \( \text{max}2 \) be \( \infty \); otherwise let \( \text{max}2 \) be \( \text{max}-1 \).
   d. Call \( \text{RepeatMatcher}(m, \text{min}2, \text{max}2, \text{greedy}, y, c, \text{parenIndex}, \text{parenCount}) \) and return its result.
3. Let \( \text{cap} \) be a fresh copy of \( x \)'s captures List.
4. For each integer \( k \) that satisfies \( \text{parenIndex} < k \) and \( k \leq \text{parenIndex} + \text{parenCount} \), set \( \text{cap}[k] \) to undefined.
5. Let $e$ be $x$'s `endIndex`.
6. Let $xr$ be the State $(e, cap)$.
7. If $min$ is not zero, return $m(xr, d)$.
8. If `greedy` is `false`, then
   a. Call $c(x)$ and let $z$ be its result.
   b. If $z$ is not `failure`, return $z$.
   c. Call $m(xr, d)$ and return its result.
9. Call $m(xr, d)$ and let $z$ be its result.
10. If $z$ is not `failure`, return $z$.
11. Call $c(x)$ and return its result.

**NOTE 1**

An `Atom` followed by a `Quantifier` is repeated the number of times specified by the `Quantifier`. A `Quantifier` can be non-greedy, in which case the `Atom` pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the `Atom` pattern is repeated as many times as possible while still matching the sequel. The `Atom` pattern is repeated rather than the input character sequence that it matches, so different repetitions of the `Atom` can match different input substrings.

**NOTE 2**

If the `Atom` and the sequel of the regular expression all have choice points, the `Atom` is first matched as many (or as few, if non-greedy) times as possible. All choices in the sequel are tried before moving on to the next choice in the last repetition of `Atom`. All choices in the last ($n^{th}$) repetition of `Atom` are tried before moving on to the next choice in the next-to-last ($n-1)^{th}$ 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 ($n-1)^{th}$ repetition of `Atom` and so on.

Compare

```
/a[a-z]{2,4}/.exec("abcdefgхи")
```

which returns "abcdefgхи"

```
/a[a-z]{2,4}?/.exec("abcdefgхи")
```

which returns "abc".

Consider also

```
/(aa|aabaac|ba|b|c)*.exec("aabaac")
```

which, by the choice point ordering above, returns the array

`["aaba", "ba"]`
and not any of:

```plaintext
["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:

```plaintext
"aaaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/, "$1")
```

which returns the gcd in unary notation "aaaaa".

**NOTE 3**

Step 4 of the RepeatMatcher clears Atom's captures each time Atom is repeated. We can see its behaviour in the regular expression

```plaintext
/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

which returns the array

```plaintext
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```plaintext
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost * clears all captured Strings contained in the quantified Atom, which in this case includes capture Strings numbered 2, 3, 4, and 5.

**NOTE 4**

Step 1 of the RepeatMatcher's d closure states that, once the minimum number of repetitions has been satisfied, any more expansions of Atom that match the empty character sequence are not considered for further repetitions. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```plaintext
/(a*)*/.exec("b")
```

or the slightly more complicated:

```plaintext
/(a*)b\1+/\.exec("baaaac")
```

which returns the array

```plaintext
["b", "]
```

### 21.2.2.6 Assertion
The production `Assertion :: ^` evaluates as follows:

1. Return an internal AssertionTester closure that takes a State argument `x` and performs the following steps when evaluated:
   a. Let `e` be `x`'s `endIndex`.
   b. If `e` is zero, return `true`.
   c. If `Multiline` is `false`, return `false`.
   d. If the character `Input[e-1]` is one of `LineTerminator`, return `true`.
   e. Return `false`.

**NOTE** Even when the `y` flag is used with a pattern, `^` always matches only at the beginning of `Input`, or (if `Multiline` is `true`) at the beginning of a line.

The production `Assertion :: $` evaluates as follows:

1. Return an internal AssertionTester closure that takes a State argument `x` and performs the following steps when evaluated:
   a. Let `e` be `x`'s `endIndex`.
   b. If `e` is equal to `InputLength`, return `true`.
   c. If `Multiline` is `false`, return `false`.
   d. If the character `Input[e]` is one of `LineTerminator`, return `true`.
   e. Return `false`.

The production `Assertion :: \ b` evaluates as follows:

1. Return an internal AssertionTester closure that takes a State argument `x` and performs the following steps when evaluated:
   a. Let `e` be `x`'s `endIndex`.
   b. Call `IsWordChar(e-1)` and let `a` be the Boolean result.
   c. Call `IsWordChar(e)` and let `b` be the Boolean result.
   d. If `a` is `true` and `b` is `false`, return `true`.
   e. If `a` is `false` and `b` is `true`, return `true`.
   f. Return `false`.

The production `Assertion :: \ B` evaluates as follows:

1. Return an internal AssertionTester closure that takes a State argument `x` and performs the following steps when evaluated:
   a. Let `e` be `x`'s `endIndex`.
   b. Call `IsWordChar(e-1)` and let `a` be the Boolean result.
   c. Call `IsWordChar(e)` and let `b` be the Boolean result.
   d. If `a` is `true` and `b` is `false`, return `false`.
   e. If `a` is `false` and `b` is `true`, return `false`. 

The production Assertion :: ( ? = Disjunction ) evaluates as follows:

1. Evaluate Disjunction to obtain a Matcher m.
2. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following steps:
   a. Let d be a Continuation that always returns its State argument as a successful MatchResult.
   b. Call m(x, d) and let r be its result.
   c. If r is failure, return failure.
   d. Let y be r’s State.
   e. Let cap be y's captures List.
   f. Let xe be x’s endIndex.
   g. Let z be the State (xe, cap).
   h. Call c(z) and return its result.

The production Assertion :: ( ? ! Disjunction ) evaluates as follows:

1. Evaluate Disjunction to obtain a Matcher m.
2. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following steps:
   a. Let d be a Continuation that always returns its State argument as a successful MatchResult.
   b. Call m(x, d) and let r be its result.
   c. If r is not failure, return failure.
   d. Call c(x) and return its result.

21.2.2.6.1 Runtime Semantics: WordCharacters ()

The abstract operation WordCharacters performs the following steps:

1. Let A be a set of characters containing the sixty-three characters:

   a b c d e f g h i j k l m n o p q r s t u v w x y z
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   0 1 2 3 4 5 6 7 8 9 _

2. Let U be an empty set.
3. For each character c not in set A where Canonicalize(c) is in A, add c to U.
4. Assert: Unless Unicode and IgnoreCase are both true, U is empty.
5. Add the characters in set U to set A.
6. Return A.
21.2.2.6.2 Runtime Semantics: IsWordChar ( e )

The abstract operation IsWordChar takes an integer parameter e and performs the following steps:

1. If e is -1 or e is InputLength, return false.
2. Let c be the character Input[e].
3. Let wordChars be the result of !WordCharacters().
4. If c is in wordChars, return true.
5. Return false.

21.2.2.7 Quantifier

The production Quantifier :: QuantifierPrefix evaluates as follows:

1. Evaluate QuantifierPrefix to obtain the two results: an integer min and an integer (or $\infty$) max.
2. Return the three results min, max, and true.

The production Quantifier :: QuantifierPrefix ? evaluates as follows:

1. Evaluate QuantifierPrefix to obtain the two results: an integer min and an integer (or $\infty$) max.
2. Return the three results min, max, and false.

The production QuantifierPrefix :: * evaluates as follows:

1. Return the two results 0 and $\infty$.

The production QuantifierPrefix :: + evaluates as follows:

1. Return the two results 1 and $\infty$.

The production QuantifierPrefix :: ? evaluates as follows:

1. Return the two results 0 and 1.

The production QuantifierPrefix :: { DecimalDigits } evaluates as follows:

1. Let i be the MV of DecimalDigits (see 11.8.3).
2. Return the two results i and i.

The production QuantifierPrefix :: { DecimalDigits , } evaluates as follows:

1. Let i be the MV of DecimalDigits.
2. Return the two results i and $\infty$.

The production QuantifierPrefix :: { DecimalDigits , DecimalDigits } evaluates as follows:
1. Let $i$ be the MV of the first $DecimalDigits$.
2. Let $j$ be the MV of the second $DecimalDigits$.
3. Return the two results $i$ and $j$.

21.2.2.8 Atom

The production $Atom :: PatternCharacter$ evaluates as follows:

1. Let $ch$ be the character matched by $PatternCharacter$.
2. Let $A$ be a one-element CharSet containing the character $ch$.
3. Call $CharacterSetMatcher(A, false)$ and return its Matcher result.

The production $Atom ::$ evaluates as follows:

1. Let $A$ be the set of all characters except $LineTerminator$.
2. Call $CharacterSetMatcher(A, false)$ and return its Matcher result.

The production $Atom :: \ AtomEscape$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $AtomEscape$.

The production $Atom :: CharacterClass$ evaluates as follows:

1. Evaluate $CharacterClass$ to obtain a CharSet $A$ and a Boolean $invert$.
2. Call $CharacterSetMatcher(A, invert)$ and return its Matcher result.

The production $Atom :: ( Disjunction )$ evaluates as follows:

1. Evaluate $Disjunction$ to obtain a Matcher $m$.
2. Let $parenIndex$ be the number of left-capturing parentheses in the entire regular expression that occur to the left of this $Atom$. This is the total number of $Atom :: ( Disjunction )$ Parse Nodes prior to or enclosing this $Atom$.
3. Return an internal Matcher closure that takes two arguments, a State $x$ and a Continuation $c$, and performs the following steps:
   a. Let $d$ be an internal Continuation closure that takes one State argument $y$ and performs the following steps:
      i. Let $cap$ be a fresh copy of $y$'s captures List.
      ii. Let $xe$ be $x$'s endIndex.
      iii. Let $ye$ be $y$'s endIndex.
      iv. Let $s$ be a fresh List whose characters are the characters of $Input$ at indices $xe$ (inclusive) through $ye$ (exclusive).
      v. Set $cap[parenIndex+1]$ to $s$.
      vi. Let $z$ be the State $(ye, cap)$. 
vii. Call \( c(z) \) and return its result.

b. Call \( m(x, d) \) and return its result.

The production \( \text{Atom} :: ( \ ? : \text{Disjunction} ) \) evaluates as follows:

1. Return the Matcher that is the result of evaluating \( \text{Disjunction} \).

**21.2.2.8.1 Runtime Semantics: CharacterSetMatcher ( \( A, \text{invert} \) )**

The abstract operation CharacterSetMatcher takes two arguments, a CharSet \( A \) and a Boolean flag \( \text{invert} \), and performs the following steps:

1. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following steps when evaluated:
   a. Let \( e \) be \( x \)'s \( \text{endIndex} \).
   b. If \( e \) is \( \text{InputLength} \), return \( \text{failure} \).
   c. Let \( ch \) be the character \( \text{Input}[e] \).
   d. Let \( cc \) be \( \text{Canonicalize}(ch) \).
   e. If \( \text{invert} \) is \( \text{false} \), then
      i. If there does not exist a member \( a \) of set \( A \) such that \( \text{Canonicalize}(a) \) is \( cc \), return \( \text{failure} \).
   f. Else \( \text{invert} \) is \( \text{true} \),
      i. If there exists a member \( a \) of set \( A \) such that \( \text{Canonicalize}(a) \) is \( cc \), return \( \text{failure} \).
   g. Let \( cap \) be \( x \)'s \( \text{captures List} \).
   h. Let \( y \) be the State \( (e+1, \text{cap}) \).
   i. Call \( c(y) \) and return its result.

**21.2.2.8.2 Runtime Semantics: Canonicalize ( \( ch \) )**

The abstract operation Canonicalize takes a character parameter \( ch \) and performs the following steps:

1. If \( \text{IgnoreCase} \) is \( \text{false} \), return \( ch \).
2. If \( \text{Unicode} \) is \( \text{true} \), then
   a. If the file Case Folding.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 \( ch \).
3. Else,
   a. Assert: \( ch \) is a UTF-16 code unit.
   b. Let \( s \) be the ECMAScript String value consisting of the single code unit \( ch \).
   c. Let \( u \) be the same result produced as if by performing the algorithm for
      \( \text{String.prototype.toUpperCase} \) using \( s \) as the \text{this} value.
   d. Assert: \( u \) is a String value.
   e. If \( u \) does not consist of a single code unit, return \( ch \).
f. Let \( cu \) be \( u \)'s single code unit element.
g. If \( ch \)'s code unit value \( \geq 128 \) and \( cu \)'s code unit value \( < 128 \), return \( ch \).
h. Return \( cu \).

**NOTE 1**
Parentheses of the form ( \textit{Disjunction} ) serve both to group the components of the \textit{Disjunction} pattern together and to save the result of the match. The result can be used either in a backreference (\ back followed by a nonzero decimal number), referenced in a replace String, or returned as part of an array from the regular expression matching internal procedure. To inhibit the capturing behaviour of parentheses, use the form (?: \textit{Disjunction} ) instead.

**NOTE 2**
The form (\ ?= \textit{Disjunction} ) specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside \textit{Disjunction} must match at the current position, but the current position is not advanced before matching the sequel. If \textit{Disjunction} can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a (\ ?= form (this unusual behaviour is inherited from Perl). This only matters when the \textit{Disjunction} contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

\[
/(?=\text{(a+)})/\text{.exec("baabac")}
\]

matches the empty String immediately after the first \texttt{b} and therefore returns the array:

\[
["", "aaa"]
\]

To illustrate the lack of backtracking into the lookahead, consider:

\[
/(?=\text{(a+)})\text{a*b}\text{\1}/\text{.exec("baabac")}
\]

This expression returns

\[
["aba", "a"]
\]

and not:

\[
["aaaba", "a"]
\]

**NOTE 3**
The form (\ ?! \textit{Disjunction} ) specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside \textit{Disjunction} must fail to match at the current position. The current position is not advanced before matching the sequel. \textit{Disjunction} can contain capturing parentheses, but backreferences to them only make sense from within \textit{Disjunction} itself. Backreferences to these capturing parentheses from elsewhere in the
pattern always return undefined because the negative lookahead must fail for the pattern to succeed. For example,

`/(.*?)a(?!(a+)b\2c)\2(.*)/.exec("baaabaac")`

looks for an a not immediately followed by some positive number n of a's, a b, another n a's (specified by the first \2) and a c. The second \2 is outside the negative lookahead, so it matches against undefined and therefore always succeeds. The whole expression returns the array:

```
[ "baaabaac", "ba", undefined, "abaac"]
```

NOTE 4 In case-insignificant matches when Unicode is true, all characters are implicitly case-folded using the simple mapping provided by the Unicode standard immediately before they are compared. The simple mapping always maps to a single code point, so it does not map, for example, "ß" (U+00DF) to "SS". It may however map a code point outside the Basic Latin range to a character within, for example, "Γ" (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`.

### 21.2.2.9 AtomEscape

The production `AtomEscape :: DecimalEscape` evaluates as follows:

1. Evaluate `DecimalEscape` to obtain an integer `n`.
2. If `n>NcapturingParens`, throw a `SyntaxError` exception.
3. Return an internal Matcher closure that takes two arguments, a State `x` and a Continuation `c`, and performs the following steps:
   a. Let `cap` be `x`'s captures List.
   b. Let `s` be `cap[n]`.
   c. If `s` is undefined, return `c(x)`.
   d. Let `e` be `x`'s endIndex.
   e. Let `len` be `s`'s length.
   f. Let `f` be `e+len`.
   g. If `f>InputLength`, return failure.
   h. If there exists an integer `i` between 0 (inclusive) and `len` (exclusive) such that `Canonicalize(s[i])` is not the same character value as `Canonicalize(Input[e+i])`, return failure.
   i. Let `y` be the State `(f, cap)`.
   j. Call `c(y)` and return its result.

The production `AtomEscape :: CharacterEscape` evaluates as follows:
1. Evaluate `CharacterEscape` to obtain a character `ch`.
2. Let `A` be a one-element CharSet containing the character `ch`.
3. Call `CharacterSetMatcher(A, false)` and return its Matcher result.

The production `AtomEscape :: CharacterClassEscape` evaluates as follows:

1. Evaluate `CharacterClassEscape` to obtain a CharSet `A`.
2. Call `CharacterSetMatcher(A, false)` and return its Matcher result.

NOTE An escape sequence of the form `\` followed by a nonzero decimal number `n` matches the result of the `n`th set of capturing parentheses (21.2.2.1). It is an error if the regular expression has fewer than `n` capturing parentheses. If the regular expression has `n` or more capturing parentheses but the `n`th one is `undefined` because it has not captured anything, then the backreference always succeeds.

### 21.2.2.10 CharacterEscape

The production `CharacterEscape :: \0` evaluates as follows:

1. Return the character `U+0000` (NULL).

NOTE `\0` represents the `<NUL>` character and cannot be followed by a decimal digit.

The production `CharacterEscape :: ControlEscape` evaluates as follows:

1. Return the character according to Table 48.

<table>
<thead>
<tr>
<th>ControlEscape</th>
<th>Character Value</th>
<th>Code Point</th>
<th>Unicode Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>9</td>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>U+000A</td>
<td>LINE FEED (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>

The production `CharacterEscape :: c ControlLetter` evaluates as follows:

1. Let `ch` be the character matched by `ControlLetter`.
2. Let `i` be `ch`'s character value.
3. Let \( j \) be the remainder of dividing \( i \) by 32.
4. Return the character whose character value is \( j \).

The production \( \text{CharacterEscape} :: \text{HexEscapeSequence} \) evaluates as follows:

1. Return the character whose code is the SV of \( \text{HexEscapeSequence} \).

The production \( \text{CharacterEscape} :: \text{RegExpUnicodeEscapeSequence} \) evaluates as follows:

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

The production \( \text{CharacterEscape} :: \text{IdentityEscape} \) evaluates as follows:

1. Return the character matched by \( \text{IdentityEscape} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u LeadSurrogate} \text{u TrailSurrogate} \) evaluates as follows:

1. Let \( \text{lead} \) be the result of evaluating \( \text{LeadSurrogate} \).
2. Let \( \text{trail} \) be the result of evaluating \( \text{TrailSurrogate} \).
3. Let \( \text{cp} \) be \( \text{UTF16Decode}(\text{lead}, \text{trail}) \).
4. Return the character whose character value is \( \text{cp} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u LeadSurrogate} \) evaluates as follows:

1. Return the character whose code is the result of evaluating \( \text{LeadSurrogate} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u TrailSurrogate} \) evaluates as follows:

1. Return the character whose code is the result of evaluating \( \text{TrailSurrogate} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u NonSurrogate} \) evaluates as follows:

1. Return the character whose code is the result of evaluating \( \text{NonSurrogate} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u Hex4Digits} \) evaluates as follows:

1. Return the character whose code is the SV of \( \text{Hex4Digits} \).

The production \( \text{RegExpUnicodeEscapeSequence} :: \text{u} \{ \text{HexDigits} \} \) evaluates as follows:

1. Return the character whose code is the MV of \( \text{HexDigits} \).

The production \( \text{LeadSurrogate} :: \text{Hex4Digits} \) evaluates as follows:

1. Return the character whose code is the SV of \( \text{Hex4Digits} \).
The production \textit{TrailSurrogate} :: \textit{Hex4Digits} evaluates as follows:

1. Return the character whose code is the SV of \textit{Hex4Digits}.

The production \textit{NonSurrogate} :: \textit{Hex4Digits} evaluates as follows:

1. Return the character whose code is the SV of \textit{Hex4Digits}.

\subsection*{21.2.2.11 DecimalEscape}

The production \textit{DecimalEscape} :: \textit{NonZeroDigit} evaluates as follows:

1. Return the MV of \textit{NonZeroDigit}.

The production \textit{DecimalEscape} :: \textit{NonZeroDigit} \textit{DecimalDigits} evaluates as follows:

1. Let \( n \) be the number of code points in \textit{DecimalDigits}.
2. Return (the MV of \textit{NonZeroDigit} \( \times 10^n \)) plus the MV of \textit{DecimalDigits}.

The definitions of “the MV of \textit{NonZeroDigit}” and “the MV of \textit{DecimalDigits}” are in 11.8.3.

\begin{quote}
\textbf{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.
\end{quote}

\subsection*{21.2.2.12 CharacterClassEscape}

The production \textit{CharacterClassEscape} :: \texttt{d} evaluates as follows:

1. Return the ten-element set of characters containing the characters 0 through 9 inclusive.

The production \textit{CharacterClassEscape} :: \texttt{D} evaluates as follows:

1. Return the set of all characters not included in the set returned by \textit{CharacterClassEscape} :: \texttt{d}.

The production \textit{CharacterClassEscape} :: \texttt{s} evaluates as follows:

1. Return the set of characters containing the characters that are on the right-hand side of the \textit{WhiteSpace} or \textit{LineTerminator} productions.

The production \textit{CharacterClassEscape} :: \texttt{S} evaluates as follows:

1. Return the set of all characters not included in the set returned by \textit{CharacterClassEscape} :: \texttt{s}.

The production \textit{CharacterClassEscape} :: \texttt{w} evaluates as follows:
1. Return the set of all characters returned by `WordCharacters()`.

The production `CharacterClassEscape :: W` evaluates as follows:

1. Return the set of all characters not included in the set returned by `CharacterClassEscape :: W`.

### 21.2.2.13 CharacterClass

The production `CharacterClass :: [ ClassRanges ]` evaluates as follows:

1. Evaluate `ClassRanges` to obtain a CharSet `A`.
2. Return the two results `A` and `false`.

The production `CharacterClass :: [ ^ ClassRanges ]` evaluates as follows:

1. Evaluate `ClassRanges` to obtain a CharSet `A`.
2. Return the two results `A` and `true`.

### 21.2.2.14 ClassRanges

The production `ClassRanges :: [empty]` evaluates as follows:

1. Return the empty CharSet.

The production `ClassRanges :: NonemptyClassRanges` evaluates as follows:

1. Evaluate `NonemptyClassRanges` to obtain a CharSet `A`.
2. Return `A`.

### 21.2.2.15 NonemptyClassRanges

The production `NonemptyClassRanges :: ClassAtom` evaluates as follows:

1. Return the CharSet that is the result of evaluating `ClassAtom`.

The production `NonemptyClassRanges :: ClassAtom NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtom` to obtain a CharSet `A`.
2. Evaluate `NonemptyClassRangesNoDash` to obtain a CharSet `B`.
3. Return the union of CharSets `A` and `B`.

The production `NonemptyClassRanges :: ClassAtom - ClassAtom ClassRanges` evaluates as follows:

1. Evaluate the first `ClassAtom` to obtain a CharSet `A`.
2. Evaluate the second `ClassAtom` to obtain a CharSet `B`.
3. Evaluate `ClassRanges` to obtain a CharSet `C`.
4. Call `CharacterRange(A, B)` and let `D` be the resulting CharSet.
5. Return the union of CharSets `D` and `C`.

### 21.2.2.15.1 Runtime Semantics: CharacterRange (A, B)

The abstract operation CharacterRange takes two CharSet parameters `A` and `B` and performs the following steps:

1. If `A` does not contain exactly one character or `B` does not contain exactly one character, throw a `SyntaxError` exception.
2. Let `a` be the one character in CharSet `A`.
3. Let `b` be the one character in CharSet `B`.
4. Let `i` be the character value of character `a`.
5. Let `j` be the character value of character `b`.
6. If `i > j`, throw a `SyntaxError` exception.
7. Return the set containing all characters numbered `i` through `j`, inclusive.

### 21.2.2.16 NonemptyClassRangesNoDash

The production `NonemptyClassRangesNoDash :: ClassAtom` evaluates as follows:

1. Return the CharSet that is the result of evaluating `ClassAtom`.

The production `NonemptyClassRangesNoDash :: ClassAtomNoDash NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet `A`.
2. Evaluate `NonemptyClassRangesNoDash` to obtain a CharSet `B`.
3. Return the union of CharSets `A` and `B`.

The production `NonemptyClassRangesNoDash :: ClassAtomNoDash - ClassAtom ClassRanges` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet `A`.
2. Evaluate `ClassAtom` to obtain a CharSet `B`.
3. Evaluate `ClassRanges` to obtain a CharSet `C`.
4. Call `CharacterRange(A, B)` and let `D` be the resulting CharSet.
5. Return the union of CharSets `D` and `C`.

**NOTE 1** `ClassRanges` can expand into a single `ClassAtom` and/or ranges of two `ClassAtom` separated by dashes. In the latter case the `ClassRanges` includes all characters between the first `ClassAtom` and the second `ClassAtom`, inclusive; an error occurs if either `ClassAtom` does not
represent a single character (for example, if one is \w) or if the first ClassAtom's character value is greater than the second ClassAtom's character value.

**NOTE 2**

Even if the pattern ignores case, the case of the two ends of a range is significant in determining which characters belong to the range. Thus, for example, the pattern /[E-F]/i matches only the letters E, F, e, and f, while the pattern /[E-f]/i matches all upper and lower-case letters in the Unicode Basic Latin block as well as the symbols [, \, ], ^, _, and `.

**NOTE 3**

A - character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of ClassRanges, the beginning or end limit of a range specification, or immediately follows a range specification.

### 21.2.2.17 ClassAtom

The production ClassAtom :: - evaluates as follows:

1. Return the CharSet containing the one character -.

The production ClassAtom :: ClassAtomNoDash evaluates as follows:

1. Evaluate ClassAtomNoDash to obtain a CharSet A.
2. Return A.

### 21.2.2.18 ClassAtomNoDash

The production ClassAtomNoDash :: SourceCharacter but not one of \ or ] or - evaluates as follows:

1. Return the CharSet containing the character matched by SourceCharacter.

The production ClassAtomNoDash :: \ ClassEscape evaluates as follows:

1. Return the CharSet that is the result of evaluating ClassEscape.

### 21.2.2.19 ClassEscape

The production ClassEscape :: b evaluates as follows:

1. Return the CharSet containing the single character <BS> U+0008 (BACKSPACE).

The production ClassEscape :: - evaluates as follows:

1. Return the CharSet containing the single character - U+002D (HYPHEN-MINUS).

The production ClassEscape :: CharacterEscape evaluates as follows:
1. Return the CharSet containing the single character that is the result of evaluating CharacterEscape.

The production ClassEscape :: CharacterClassEscape evaluates as follows:

1. Return the CharSet that is the result of evaluating CharacterClassEscape.

NOTE A ClassAtom can use any of the escape sequences that are allowed in the rest of the regular expression except for \b, \B, and backreferences. Inside a CharacterClass, \b means the backspace character, while \B and backreferences raise errors. Using a backreference inside a ClassAtom causes an error.

21.2.3 The RegExp Constructor

The RegExp constructor is the %RegExp% intrinsic object and the initial value of the RegExp property of the global object. When RegExp is called as a function rather than as a constructor, it creates and initializes a new RegExp object. Thus the function call RegExp(...) is equivalent to the object creation expression new RegExp(...) with the same arguments.

The RegExp 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 RegExp behaviour must include a super call to the RegExp constructor to create and initialize subclass instances with the necessary internal slots.

21.2.3.1 RegExp (pattern, flags)

The following steps are taken:

2. If NewTarget is not undefined, let newTarget be NewTarget.
3. Else,
   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.
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 \( \text{flags} \).

6. Else,
   a. Let \( P \) be \( \text{pattern} \).
   b. Let \( F \) be \( \text{flags} \).

7. Let \( O \) be ? \( \text{RegExpAlloc} \( \text{newTarget} \) \).

8. Return ? \( \text{RegExpInitialize} \( O, P, F \) \).

**NOTE**

If pattern is supplied using a \( \text{StringLiteral} \), 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 \( \text{StringLiteral} \) to prevent them being removed when the contents of the \( \text{StringLiteral} \) are formed.

### 21.2.3.2 Abstract Operations for the RegExp Constructor

#### 21.2.3.2.1 Runtime Semantics: RegExpAlloc (newTarget)

When the abstract operation RegExpAlloc with argument \( \text{newTarget} \) is called, the following steps are taken:

1. Let \( \text{obj} \) be ? \( \text{OrdinaryCreateFromConstructor} \( \text{newTarget} \), "%RegExpPrototype%", «
   [[RegExpMatcher]], [[OriginalSource]], [[OriginalFlags]] »).
2. Perform ! \( \text{DefinePropertyOrThrow} \( \text{obj} \), "lastIndex"", PropertyDescriptor {[[Writable]]: \text{true},
   [[Enumerable]]: \text{false}, [[Configurable]]: \text{false}}).
3. Return \( \text{obj} \).

#### 21.2.3.2.2 Runtime Semantics: RegExpInitialize (obj, pattern, flags)

When the abstract operation RegExpInitialize with arguments \( \text{obj} \), \( \text{pattern} \), and \( \text{flags} \) is called, the following steps are taken:

1. If \( \text{pattern} \) is \text{undefined}, let \( \text{P} \) be the empty String.
2. Else, let \( \text{P} \) be ? \( \text{ToString} \( \text{pattern} \) \).
3. If \( \text{flags} \) is \text{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\", \"u\", or \"y\" or if it contains the same code unit more than once, throw a \text{SyntaxError} exception.
6. If \( \text{F} \) contains \"u\", let \( \text{BMP} \) be \text{false}; else let \( \text{BMP} \) be \text{true}.
7. If \( \text{BMP} \) is \text{true}, then
   a. Parse \( \text{P} \) using the grammars in 21.2.1 and interpreting each of its 16-bit elements as a Unicode BMP code point. UTF-16 decoding is not applied to the elements. The \text{goal symbol} for the parse...
is $\text{Pattern}[^{-}\text{U}]$. Throw a SyntaxError exception if $P$ did not conform to the grammar, if any elements of $P$ were not matched by the parse, or if any Early Error conditions exist.

b. Let $\text{patternCharacters}$ be a List whose elements are the code unit elements of $P$.

8. Else,
   a. Parse $P$ using the grammars in 21.2.1 and interpreting $P$ as UTF-16 encoded Unicode code points (6.1.4). The goal symbol for the parse is $\text{Pattern}[^{+}\text{U}]$. Throw a SyntaxError exception if $P$ did not conform to the grammar, if any elements of $P$ were not matched by the parse, or if any Early Error conditions exist.
   b. Let $\text{patternCharacters}$ be a List whose elements are the code points resulting from applying UTF-16 decoding to $P$’s sequence of elements.

9. Set $\text{obj}[[\text{OriginalSource}]]$ to $P$.
10. Set $\text{obj}[[\text{OriginalFlags}]]$ to $F$.
11. Set $\text{obj}[[\text{RegExpMatcher}]]$ to the internal procedure that evaluates the above parse of $P$ by applying the semantics provided in 21.2.2 using $\text{patternCharacters}$ as the pattern’s List of SourceCharacter values and $F$ as the flag parameters.
12. Perform $\text{Set}(\text{obj}, \text{"lastIndex"}, 0, \text{true})$.
13. Return $\text{obj}$.

### 21.2.3.2.3 Runtime Semantics: RegExpCreate ($P$, $F$)

When the abstract operation RegExpCreate with arguments $P$ and $F$ is called, the following steps are taken:

1. Let $\text{obj}$ be $\text{RegExpAlloc}(%\text{RegExp}%)$.
2. Return $\text{RegExpInitialize}(\text{obj}, P, F)$.

### 21.2.3.2.4 Runtime Semantics: EscapeRegExpPattern ($P$, $F$)

When the abstract operation EscapeRegExpPattern with arguments $P$ and $F$ is called, the following occurs:

1. Let $S$ be a String in the form of a $\text{Pattern}[^{-}\text{U}]$ ($\text{Pattern}[^{+}\text{U}]$ 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 internal procedure that would result from evaluating $S$ as a $\text{Pattern}[^{-}\text{U}]$ ($\text{Pattern}[^{+}\text{U}]$ if $F$ contains "u") must behave identically to the internal procedure given by the constructed object’s [[RegExpMatcher]] internal slot. Multiple calls to this abstract operation using the same values for $P$ and $F$ must produce identical results.
2. The code points / or any LineTerminator occurring in the pattern shall be escaped in $S$ as necessary to ensure that the String value formed by concatenating the Strings "/", $S$, "/", and $F$ can be parsed (in an appropriate lexical context) as a RegularExpressionLiteral that behaves identically to the constructed regular expression. For example, if $P$ is "/", then $S$ could be "/" or "\u002F", among other possibilities, but not "/", because "/" followed by $F$ would be parsed as a SingleLineComment
rather than a `RegularExpressionLiteral`. If `P` is the empty String, this specification can be met by letting `S` be `"(?:)".

3. Return `S`.

### 21.2.4 Properties of the RegExp Constructor

The value of the `[[Prototype]]` internal slot of the RegExp constructor is the intrinsic object `%FunctionPrototype%`.

The RegExp constructor has the following properties:

#### 21.2.4.1 `RegExp.prototype`

The initial value of `RegExp.prototype` is the intrinsic object `%RegExpPrototype%`.

This property has the attributes `{ `[[Writable]]: `false, `[[Enumerable]]: `false, `[[Configurable]]: `false }.

#### 21.2.4.2 `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` object’s constructor to create a derived object. However, a subclass constructor may over-ride that default behaviour by redefining its `@@species` property.

### 21.2.5 Properties of the RegExp Prototype Object

The RegExp prototype object is the intrinsic object `%RegExpPrototype%`. The RegExp prototype object is an ordinary object. It is not a RegExp instance and does not have a `[[RegExpMatcher]]` internal slot or any of the other internal slots of RegExp instance objects.

The value of the `[[Prototype]]` internal slot of the RegExp prototype object is the intrinsic object `%ObjectPrototype%`.

**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.
21.2.5.1  RegExp.prototype.constructor

The initial value of `RegExp.prototype.constructor` is the intrinsic object `%RegExp%`.

21.2.5.2  RegExp.prototype.exec ( `string` )

Performs a regular expression match of `string` against the regular expression and returns an Array object containing the results of the match, or null if `string` did not match.

The String `toString(string)` is searched for an occurrence of the regular expression pattern as follows:

1. Let `R` be the `this` value.
2. If `Type(R)` is not Object, throw a `TypeError` exception.
3. If `R` does not have a `[[RegExpMatcher]]` internal slot, throw a `TypeError` exception.
4. Let `S` be `toString(string)`.
5. Return `RegExpBuiltinExec(R, S)`.

21.2.5.2.1  Runtime Semantics: RegExpExec ( `R, S` )

The abstract operation `RegExpExec` with arguments `R` and `S` performs the following steps:

1. Assert: `Type(R)` is Object.
2. Assert: `Type(S)` is String.
3. Let `exec` be `Get(R, "exec")`.
4. If `IsCallable(exec)` is `true`, then
   a. Let `result` be `Call(exec, R, « S »)`.
   b. If `Type(result)` is neither Object or Null, throw a `TypeError` exception.
   c. Return `result`.
5. If `R` does not have a `[[RegExpMatcher]]` internal slot, throw a `TypeError` exception.

**NOTE** If a callable `exec` property is not found this algorithm falls back to attempting to use the built-in RegExp matching algorithm. This provides compatible behaviour for code written for prior editions where most built-in algorithms that use regular expressions did not perform a dynamic property lookup of `exec`.

21.2.5.2.2  Runtime Semantics: RegExpBuiltinExec ( `R, S` )

The abstract operation `RegExpBuiltinExec` with arguments `R` and `S` performs the following steps:

1. Assert: `R` is an initialized RegExp instance.
2. Assert: `Type(S)` is String.
3. Let `length` be the number of code units in `S`. 
4. Let `lastIndex` be \( \text{ToLength(? Get(R, "lastIndex"))} \).

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.

9. Let `matcher` be \( R[[\text{RegExpMatcher}]] \).

10. If `flags` contains "u", let `fullUnicode` be `true`, else let `fullUnicode` be `false`.

11. Let `matchSucceeded` be `false`.

12. Repeat, while `matchSucceeded` is `false`

   a. If `lastIndex > length`, then
      i. If `global` is `true` or `sticky` is `true`, then
         1. Perform `Set(R, "lastIndex", 0, true)`.
         2. Return `null`.
      b. Let `r` be `matcher(S, lastIndex)`.
      c. If `r` is failure, then
         i. If `sticky` is `true`, then
            1. Perform `Set(R, "lastIndex", 0, true)`.
            2. Return `null`.
         ii. Set `lastIndex` to `AdvanceStringIndex(S, lastIndex, fullUnicode)`.
      d. Else,
         i. Assert: `r` is a State.
         ii. Set `matchSucceeded` to `true`.

13. Let `e` be `r`'s `endIndex` value.

14. If `fullUnicode` is `true`, then

   a. `e` is an index into the `Input` character list, derived from `S`, matched by `matcher`. Let `eUTF` be the smallest index into `S` that corresponds to the character at element `e` of `Input`. If `e` is greater than or equal to the length of `Input`, then `eUTF` is the number of code units in `S`.

   b. Set `e` to `eUTF`.

15. If `global` is `true` or `sticky` is `true`, then

   a. Perform `Set(R, "lastIndex", e, true)`.

16. Let `n` be the length of `r`'s `captures List`. (This is the same value as 21.2.2.1's `NcapturingParen`s.)

17. Let `A` be `ArrayCreate(n + 1)`.

18. Assert: The value of `A`'s "length" property is `n + 1`.

19. Let `matchIndex` be `lastIndex`.

20. Perform `CreateDataProperty(A, "index", matchIndex)`.


22. Let `matchedSubstr` be the matched substring (i.e. the portion of `S` between offset `lastIndex` inclusive and offset `e` exclusive).

23. Perform `CreateDataProperty(A, "\0", matchedSubstr)`.

24. For each integer `i` such that `i > 0` and `i ≤ n`, do
a. Let \( capture \) be \( i \)th element of \( r \)'s \( captures \) List.
b. If \( capture \) is \texttt{undefined}, let \( capturedValue \) be \texttt{undefined}.
c. Else if \( fullUnicode \) is \texttt{true}, then
   i. Assert: \( capture \) is a List of code points.
   ii. Let \( capturedValue \) be a String value whose code units are the UTF16Encoding of the code points of \( capture \).
d. Else \( fullUnicode \) is \texttt{false},
   i. Assert: \( capture \) is a List of code units.
   ii. Let \( capturedValue \) be a String consisting of the code units of \( capture \).
e. Perform \( ! \text{CreateDataProperty}(A, !\text{ToString}(i), capturedValue) \).

25. Return \( A \).

21.2.5.2.3 \texttt{AdvanceStringLength(} \( S \), \texttt{index}, \( unicode \) \texttt{)}

The abstract operation \texttt{AdvanceStringLength} with arguments \( S \), \( index \), and \( unicode \) performs the following steps:

1. Assert: Type\((S)\) is String.
2. Assert: \( index \) is an integer such that \( 0 \leq index \leq 2^{53} - 1 \).
3. Assert: Type\((unicode)\) is Boolean.
4. If \( unicode \) is \texttt{false}, return \( index+1 \).
5. Let \( length \) be the number of code units in \( S \).
6. If \( index+1 \geq length \), return \( index+1 \).
7. Let \( first \) be the code unit value at index \( index \) in \( S \).
8. If \( first < 0x\text{D}800 \) or \( first > 0xB\text{FF} \), return \( index+1 \).
9. Let \( second \) be the code unit value at index \( index+1 \) in \( S \).
10. If \( second < 0xC\text{D}00 \) or \( second > 0xD\text{FF} \), return \( index+1 \).
11. Return \( index+2 \).

21.2.5.3 get \texttt{RegExp.prototype.flags}

\texttt{RegExp.prototype.flags} is an accessor property whose set accessor function is \texttt{undefined}. Its get accessor function performs the following steps:

1. Let \( R \) be the \texttt{this} value.
2. If Type\((R)\) is not Object, throw a \texttt{TypeError} exception.
3. Let \( result \) be the empty String.
4. Let \( global \) be \( !\text{ToBooolean}(? \text{Get}(R, "global")) \).
5. If \( global \) is \texttt{true}, append "g" as the last code unit of \( result \).
6. Let \( ignoreCase \) be \( !\text{ToBooolean}(? \text{Get}(R, "ignoreCase")) \).
7. If \( ignoreCase \) is \texttt{true}, append "i" as the last code unit of \( result \).
8. Let `multiline` be `ToBoolean(? Get(R, "multiline"))`.
9. If `multiline` is `true`, append "m" as the last code unit of `result`.
10. Let `unicode` be `ToBoolean(? Get(R, "unicode"))`.
11. If `unicode` is `true`, append "u" as the last code unit of `result`.
12. Let `sticky` be `ToBoolean(? Get(R, "sticky"))`.
13. If `sticky` is `true`, append "y" as the last code unit of `result`.
14. Return `result`.

21.2.5.4 `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, %RegExpPrototype%)` is `true`, return `undefined`.
   b. Otherwise, throw a `TypeError` exception.
4. Let `flags` be `R.[[OriginalFlags]]`.
5. If `flags` contains the code unit "g", return `true`.
6. Return `false`.

21.2.5.5 `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, %RegExpPrototype%)` is `true`, return `undefined`.
   b. Otherwise, throw a `TypeError` exception.
4. Let `flags` be `R.[[OriginalFlags]]`.
5. If `flags` contains the code unit "i", return `true`.
6. Return `false`.

21.2.5.6 `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 `global` is `true`,
   a. Let `fullUnicode` be `ToBoolean(? Get(rx, "unicode"))`.
   b. Perform `? Set(rx, "lastIndex", 0, true)`.
   c. Let `A` be `! ArrayCreate(0)`.
   d. Let `n` be `0`.
   e. 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 `result` is not `null`,
           1. Let `matchStr` be `? ToString(? Get(result, "\0"))`.
           2. Let `status` be `CreateDataProperty(A, ! ToString(n), matchStr)`.
           3. Assert: `status` is `true`.
           4. If `matchStr` is the empty String, then
              a. Let `thisIndex` be `? ToLength(? Get(rx, "lastIndex"))`.
              b. Let `nextIndex` be `AdvanceStringIndex(S, thisIndex, fullUnicode)`.
              c. Perform `? Set(rx, "lastIndex", nextIndex, true)`.
      5. Increment `n`.

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.

### 21.2.5.7 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, %RegExpPrototype%)` is `true`, return `undefined`.
   b. Otherwise, throw a `TypeError` exception.
4. Let `flags` be `R.OriginalFlags`.
5. If `flags` contains the code unit "m", return `true`.
6. Return `false`.

21.2.5.8 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 `Callable(replaceValue)`.
6. If `functionalReplace` is `false`, then
   a. Let `replaceValue` be `ToString(replaceValue)`.
7. Let `global` be `ToBoolean(? Get(rx, "global"))`.
8. If `global` is `true`, then
   a. Let `fullUnicode` be `ToBoolean(? Get(rx, "unicode"))`.
   b. Perform `Set(rx, "lastIndex", 0, 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 `result` is not `null`,
      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 `ToLength(? Get(rx, "lastIndex"))`.
            b. Let `nextIndex` be `AdvanceStringIndex(S, thisIndex, fullUnicode)`.
            c. Perform `Set(rx, "lastIndex", nextIndex, true)`.
12. Let `accumulatedResult` be the empty String value.
13. Let `nextSourcePosition` be 0.
14. For each `result` in `results`, do
   a. Let `nCaptures` be `ToLength(? Get(result, "length"))`.
   b. Let `nCaptures` be `max(nCaptures - 1, 0)`.
   c. Let `matched` be `ToString(? Get(result, "\0"))`.
   d. Let `matchLength` be the number of code units in `matched`. 
The value of the name property of this function is "[Symbol.replace]".

### 21.2.5.9 RegExp.prototype [ @@search ] (string)

When the @@search method is called with argument string, the following steps are taken:

1. Let \( rx \) be the this value.
2. If Type(\( rx \)) is not Object, throw a TypeError exception.
Let $S$ be $\text{ToString}(\text{string})$.

4. Let $\text{previousLastIndex}$ be $\text{Get}(\text{rx}, \text{"lastIndex"})$.

5. If $\text{SameValue}(\text{previousLastIndex}, 0)$ is $\text{false}$, then
   a. Perform $\text{Set}(\text{rx}, \text{"lastIndex"}, 0, \text{true})$.

6. Let $\text{result}$ be $\text{RegExpExec}(\text{rx}, S)$.

7. Let $\text{currentLastIndex}$ be $\text{Get}(\text{rx}, \text{"lastIndex"})$.

8. If $\text{SameValue}(\text{currentLastIndex}, \text{previousLastIndex})$ is $\text{false}$, then
   a. Perform $\text{Set}(\text{rx}, \text{"lastIndex"}, \text{previousLastIndex}, \text{true})$.

9. If $\text{result}$ is $\text{null}$, return -1.

10. Return $\text{Get}(\text{result}, \text{"index"})$.

The value of the $\text{name}$ property of this function is "$[\text{Symbol.search}]$".

**NOTE**

The $\text{lastIndex}$ and $\text{global}$ properties of this RegExp object are ignored when performing the search. The $\text{lastIndex}$ property is left unchanged.

### 21.2.5.10 get RegExp.prototype.source

$\text{RegExp.prototype.source}$ is an accessor property whose set accessor function is $\text{undefined}$. Its get accessor function performs the following steps:

1. Let $R$ be the $\text{this}$ value.
2. If $\text{Type}(R)$ is not Object, throw a $\text{TypeError}$ exception.
3. If $R$ does not have an $[[\text{OriginalSource}]]$ internal slot, then
   a. If $\text{SameValue}(R, \%\text{RegExpPrototype}\%)$ is $\text{true}$, return "$(:?)".
   b. Otherwise, throw a $\text{TypeError}$ exception.
4. Assert: $R$ has an $[[\text{OriginalFlags}]]$ internal slot.
5. Let $\text{src}$ be $R[[\text{OriginalSource}]]$.
6. Let $\text{flags}$ be $R[[\text{OriginalFlags}]]$.
7. Return $\text{EscapeRegExpPattern}(\text{src}, \text{flags})$.

### 21.2.5.11 RegExp.prototype [ @@split ] (string, limit)

**NOTE 1**

Returns an Array object into which substrings of the result of converting $\text{string}$ to a String have stored. The substrings are determined by searching from left to right for matches of the $\text{this}$ regular expression; these occurrences are not part of any substring in the returned array, but divide up the String value.

The $\text{this}$ value may be an empty regular expression or a regular expression that can match a String. In this case, the regular expression does not match the empty substring at the beginning of the input String, nor does it match the empty substring at the end of the previous separator (For example, if the regular expression matches the empty String, the String is split up into i
code unit elements; the length of the result array equals the length of the String, and each sub string contains one code unit. Only the first match at a given index of the String is considered, even backtracking could yield a non-empty-substring match at that index. (For example, `/a*/[Symbol.split]("ab")` evaluates to the array `"a","b"`, while `/a*/[Symbol.split]("ab")` evaluates to the array `"","b"`.)

If the `string` is (or converts to) the empty String, the result depends on whether the regular expression can match the empty String. If it can, the result array contains no elements. Otherwise, the result 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","","C"

If limit is not undefined, then the output array is truncated so that it contains no more than 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 that is the 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 `ToUint32(limit)`.
14. Let `size` be the number of elements in `S`.
15. Let `p` be 0.
16. If `lim` = 0, return `A`.
17. If `size` = 0, then
   a. Let `z` be `RegExpExec(splitter, S)`. 
b. If z is not null, return A.
d. Return A.

18. Let q be p.
19. Repeat, while q < size
   a. Perform ? Set(splitter, "lastIndex", q, true).
   b. Let z be ? RegExpExec(splitter, S).
   c. If z is null, let q be AdvanceStringIndex(S, q, unicodeMatching).
   d. Else z is not null,
      i. Let e be ? ToLength(? Get(splitter, "lastIndex")).
      ii. Let e be min(e, size).
      iii. If e = p, let q be AdvanceStringIndex(S, q, unicodeMatching).
      iv. Else e ≠ p,
         1. Let T be a String value equal to the substring of S consisting of the elements at
            indices p (inclusive) through q (exclusive).
         2. Perform ! CreateDataProperty(A, ! ToString(lengthA), T).
         3. Let lengthA be lengthA + 1.
         4. If lengthA = lim, return A.
         5. Let p be e.
         6. Let numberOfCaptures be ? ToLength(? Get(z, "length")).
         7. Let numberOfCaptures be max(numberOfCaptures−1, 0).
         8. Let i be 1.
         9. Repeat, while i ≤ numberOfCaptures,
            a. Let nextCapture be ? Get(z, ! ToString(i)).
            b. Perform ! CreateDataProperty(A, ! ToString(lengthA), nextCapture).
            c. Let i be i + 1.
            d. Let lengthA be lengthA + 1.
            e. If lengthA = lim, return A.
      10. Let q be p.
20. Let T be a String value equal to the substring of S consisting of the elements at indices p (inclusive)
    through size (exclusive).
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.

21.2.5.12 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, %RegExpPrototype%)` is `true`, return `undefined`.
   b. Otherwise, throw a `TypeError` exception.
4. Let `flags` be `R.[[OriginalFlags]]`.
5. If `flags` contains the code unit "y", return `true`.
6. Return `false`.

### 21.2.5.13 RegExp.prototype.test (S)

The following steps are taken:

1. Let `R` be the `this` value.
2. If `Type(R)` is not Object, throw a `TypeError` exception.
3. Let `string` be `? ToString(S)`.
4. Let `match` be `? RegExpExec(R, string)`.
5. If `match` is not `null`, return `true`; else return `false`.

### 21.2.5.14 RegExp.prototype.toString ()

1. Let `R` be the `this` value.
2. If `Type(R)` is not Object, throw a `TypeError` exception.
5. Let `result` be the String value formed by concatenating "/", `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.

### 21.2.5.15 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, %RegExpPrototype%)` is `true`, return `undefined`.
b. Otherwise, throw a `TypeError` exception.

4. Let `flags` be `R.[[OriginalFlags]]`.
5. If `flags` contains the code unit "u", return `true`.
6. Return `false`.

### 21.2.6 Properties of RegExp Instances

RegExp instances are ordinary objects that inherit properties from the RegExp prototype object. RegExp instances have internal slots `[[RegExpMatcher]]`, `[[OriginalSource]]`, and `[[OriginalFlags]]`. The value of the `[[RegExpMatcher]]` internal slot is an implementation-dependent representation of the `Pattern` of the RegExp object.

**NOTE** Prior to 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:

#### 21.2.6.1 lastIndex

The value of the `lastIndex` property specifies the String index at which to start the next match. It is coerced to an integer when used (see 21.2.5.2.2). This property shall have the attributes `{ [[Writable]]: `true`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

### 22 Indexed Collections

#### 22.1 Array Objects

Array objects are exotic objects that give special treatment to a certain class of property names. See 9.4.2 for a definition of this special treatment.

#### 22.1.1 The Array Constructor

The Array constructor is the `%Array%` intrinsic object and the initial value of the `Array` property of the `global object`. When called as a constructor it creates and initializes a new Array exotic object. When `Array` is called as a function rather than as a constructor, it also creates and initializes a new Array object. Thus
the function call `Array(...)` is equivalent to the object creation expression `new Array(...)` with the same arguments.

The `Array` constructor is a single function whose behaviour is overloaded based upon the number and types of its arguments.

The `Array` 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 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.

The `length` property of the `Array` constructor function is 1.

### 22.1.1.1 Array ()

This description applies if and only if the Array constructor is called with no arguments.

1. Let `numberOfArgs` be the number of arguments passed to this function call.
2. Assert: `numberOfArgs = 0`.
3. If `NewTarget` is `undefined`, let `newTarget` be the active function object, else let `newTarget` be `NewTarget`.
4. Let `proto` be ? `GetPrototypeFromConstructor(newTarget, "%ArrayPrototype%")`.
5. Return `! ArrayCreate(0, proto)`.

### 22.1.1.2 Array (len)

This description applies if and only if the Array constructor is called with exactly one argument.

1. Let `numberOfArgs` be the number of arguments passed to this function call.
3. If NewTarget is `undefined`, let `newTarget` be the active function object, else let `newTarget` be `NewTarget`.
4. Let `proto` be ? `GetPrototypeFromConstructor(newTarget, "%ArrayPrototype%")`.
5. Let `array` be `! ArrayCreate(0, proto)`.
6. If `Type(len)` is not Number, then
   a. Let `defineStatus` be `CreateDataProperty(array, "0", len)`.
   b. Assert: `defineStatus` is `true`.
   c. Let `intLen` be 1.
7. Else,
   a. Let `intLen` be `ToUint32(len)`.
   b. If `intLen ≠ len`, throw a `RangeError` exception.
8. Perform `Set(array, "length", intLen, true)`.  

### 22.1.3 Array (...items)

This description applies if and only if the Array constructor is called with at least two arguments.

When the `Array` function is called, the following steps are taken:

1. Let `numberOfArgs` be the number of arguments passed to this function call.  
3. If `NewTarget` is `undefined`, let `newTarget` be the active function object, else let `newTarget` be `NewTarget`.  
4. Let `proto` be `? GetPrototypeFromConstructor(newTarget, "%ArrayPrototype%")`.  
5. Let `array` be `? ArrayCreate(numberOfArgs, proto)`.  
6. Let `k` be 0.  
7. Let `items` be a zero-origined `List` containing the argument items in order.  
8. Repeat, while `k < numberOfArgs`  
   a. Let `Pk` be `! ToString(k)`.  
   b. Let `itemK` be `items[k]`.  
   c. Let `defineStatus` be `CreateDataProperty(array, Pk, itemK)`.  
   d. Assert: `defineStatus` is `true`.  
   e. Increase `k` by 1.  
9. Assert: The value of `array`'s `length` property is `numberOfArgs`.  
10. Return `array`.

### 22.1.2 Properties of the Array Constructor

The value of the `[[Prototype]]` internal slot of the Array constructor is the intrinsic object `%FunctionPrototype%`.

The Array constructor has the following properties:

### 22.1.2.1 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. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.

c. Let `mapping` be `true`.

4. Let `usingIterator` be ? `GetMethod(items, @@iterator)`.

5. If `usingIterator` is not `undefined`, then
   a. If `IsConstructor(C)` is `true`, then
      i. Let `A` be ? `Construct(C)`.
   b. Else,
      i. Let `A` be ! `ArrayCreate(0)`.
   c. Let `iterator` be ? `GetIterator(items, usingIterator)`.
   d. Let `k` be `0`.
   e. Repeat,
      i. If `k ≥ 2^{53} - 1`, then
         1. Let `error` be `Completion`([[Type]]: `throw`, [[Value]]: a newly created `TypeError` object, [[Target]]: `empty`).
         2. Return ? `IteratorClose(iterator, error)`.
      ii. Let `Pk` be ! `ToString(k)`.
      iii. Let `next` be ? `IteratorStep(iterator)`.
      iv. If `next` is `false`, then
         1. Perform ? `Set(A, "length", k, true)`.
         2. Return `A`.
   v. Let `nextValue` be ? `IteratorValue(next)`.
   vi. If `mapping` is `true`, then
      1. Let `mappedValue` be `Call(mapfn, T, « nextValue, k »)`.
      2. If `mappedValue` is an abrupt completion, return ? `IteratorClose(iterator, mappedValue)`.
      3. Let `mappedValue` be `mappedValue.[[Value]]`.
   vii. Else, let `mappedValue` be `nextValue`.
   viii. Let `defineStatus` be `CreateDataPropertyOrThrow(A, Pk, mappedValue)`.
      ix. If `defineStatus` is an abrupt completion, return ? `IteratorClose(iterator, defineStatus)`.
     x. Increase `k` by `1`.

6. NOTE: `items` is not an iterable so assume it is an array-like object.

7. Let `arrayLike` be ! `ToObject(items)`.

8. Let `len` be ? `ToLength(? `Get(arrayLike, "length")`)`.

9. If `IsConstructor(C)` is `true`, then
10. Else,
    a. Let `A` be ! `ArrayCreate(len)`.
11. Let `k` be `0`.
12. Repeat, while `k < len`
    a. Let `Pk` be ! `ToString(k)`.
b. Let $kValue$ be $\text{Get}(\text{arrayLike}, Pk)$.

c. If $\text{mapping}$ is $\text{true}$, then

   i. Let $\text{mappedValue}$ be $\text{Call}(\text{mapfn}, T, \langle kValue, k \rangle)$.

d. Else, let $\text{mappedValue}$ be $kValue$.

e. Perform $\text{CreateDataPropertyOrThrow}(A, Pk, \text{mappedValue})$.

f. Increase $k$ by 1.

13. Perform $\text{Set}(A, "length", \text{len}, \text{true})$.


NOTE  The $\text{from}$ function is an intentionally generic factory method; it does not require that its $\text{this}$ value be the Array constructor. Therefore it can be transferred to or inherited by any other constructors that may be called with a single numeric argument.

### 22.1.2.2 Array.isArray (arg)

The $\text{isArray}$ function takes one argument $\text{arg}$, and performs the following steps:

1. Return $\text{IsArray}(\text{arg})$.

### 22.1.2.3 Array.of (...items)

When the $\text{of}$ method is called with any number of arguments, the following steps are taken:

1. Let $\text{len}$ be the actual number of arguments passed to this function.
2. Let $\text{items}$ be the $\text{List}$ of arguments passed to this function.
3. Let $\text{C}$ be the $\text{this}$ value.
4. If $\text{IsConstructor}(\text{C})$ is $\text{true}$, then
   a. Let $\text{A}$ be $\text{Construct}(\text{C}, \langle \text{len} \rangle)$.

5. Else,
   a. Let $\text{A}$ be $\text{ArrayCreate}(\text{len})$.

6. Let $k$ be 0.

7. Repeat, while $k < \text{len}$
   a. Let $kValue$ be $\text{items}[k]$.
   b. Let $Pk$ be $\text{ToString}(k)$.
   c. Perform $\text{CreateDataPropertyOrThrow}(\text{A}, Pk, kValue)$.
   d. Increase $k$ by 1.

8. Perform $\text{Set}(\text{A}, "length", \text{len}, \text{true})$.

9. Return $\text{A}$.

NOTE 1  The $\text{items}$ argument is assumed to be a well-formed rest argument value.

NOTE 2  The $\text{of}$ function is an intentionally generic factory method; it does not require that its $\text{this}$
value be the Array constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument.

### 22.1.2.4 Array.prototype

The value of `Array.prototype` is `%ArrayPrototype%`, the intrinsic Array prototype object.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 22.1.2.5 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` object’s constructor to create a derived object. However, a subclass constructor may over-ride that default behaviour by redefining its @@species property.

### 22.1.3 Properties of the Array Prototype Object

The Array prototype object is the intrinsic object `%ArrayPrototype%`. The Array prototype object is an Array exotic object and has the internal methods specified for such objects. It has a `length` property whose initial value is 0 and whose attributes are `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }.

The value of the `[Prototype]` internal slot of the Array prototype object is the intrinsic object `%ObjectPrototype%`.

**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.

### 22.1.3.1 Array.prototype.concat ( ...arguments )

When the `concat` method is called with zero or more arguments, it returns an array containing the array elements of the object followed by the array elements of each argument in order.

The following steps are taken:

1. Let `O` be `?ToObject(this value)`. 

2. Let \( A \) be \( \text{ArraySpeciesCreate}(O, 0) \).
3. Let \( n \) be 0.
4. Let \( \textit{items} \) be a \texttt{List} whose first element is \( O \) and whose subsequent elements are, in left to right order, the arguments that were passed to this function invocation.
5. Repeat, while \( \textit{items} \) is not empty
   a. Remove the first element from \( \textit{items} \) and let \( E \) be the value of the element.
   b. Let \( \textit{spreadable} \) be \( \text{IsConcatSpreadable}(E) \).
   c. If \( \textit{spreadable} \) is \texttt{true}, then
      i. Let \( k \) be 0.
      ii. Let \( \textit{len} \) be \( \text{ToLength}(\text{Get}(E, "length")) \).
      iii. If \( n + \textit{len} > 2^{53} - 1 \), throw a \texttt{TypeError} exception.
   iv. Repeat, while \( k < \textit{len} \)
      1. Let \( P \) be \( \text{ToString}(k) \).
      2. Let \( \textit{exists} \) be \( \text{HasProperty}(E, P) \).
      3. If \( \textit{exists} \) is \texttt{true}, then
         a. Let \( \textit{subElement} \) be \( \text{Get}(E, P) \).
         b. Perform \( \text{CreateDataPropertyOrThrow}(A, ! \text{ToString}(n), \textit{subElement}) \).
      4. Increase \( n \) by 1.
      5. Increase \( k \) by 1.
   d. Else \( E \) is added as a single item rather than spread,
      i. If \( n \geq 2^{53} - 1 \), throw a \texttt{TypeError} exception.
      ii. Perform \( \text{CreateDataPropertyOrThrow}(A, ! \text{ToString}(n), E) \).
      iii. Increase \( n \) by 1.
6. Perform \( \text{Set}(A, "\text{length}", n, \text{true}) \).
7. Return \( A \).

The \texttt{length} property of the \texttt{concat} method is 1.

**NOTE 1** The explicit setting of the \texttt{length} property in step 6 is necessary to ensure that its value is correct in situations where the trailing elements of the result Array are not present.

**NOTE 2** The \texttt{concat} function is intentionally generic; it does not require that its \texttt{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

### 22.1.3.1.1 Runtime Semantics: \text{IsConcatSpreadable}(O)

The abstract operation \text{IsConcatSpreadable} with argument \( O \) performs the following steps:

1. If \( \text{Type}(O) \) is not \texttt{Object}, return \texttt{false}.
2. Let \( \textit{spreadable} \) be \( \text{Get}(O, @@\text{isConcatSpreadable}) \).
3. If \( \textit{spreadable} \) is not \texttt{undefined}, return \( \text{ToBoolean}(\textit{spreadable}) \).
4. Return \( \text{IsArray}(O) \).
22.1.3.2 Array.prototype.constructor

The initial value of `Array.prototype.constructor` is the intrinsic object `%Array%`.

22.1.3.3 Array.prototype.copyWithin ( `target`, `start` [ , `end` ] )

The `copyWithin` method takes up to three arguments `target`, `start` and `end`.

**NOTE 1** The `end` argument is optional with the length of the `this` object as its default value. If `target` is negative, it is treated as `length+target` where `length` is the length of the array. If `start` is negative, it is treated as `length+start`. If `end` is negative, it is treated as `length+end`.

The following steps are taken:

1. Let `O` be `ToObject(this value)`.
2. Let `len` be `ToLength(? Get(O, "length")).`
3. Let `relativeTarget` be `ToInteger(target)`.
4. If `relativeTarget < 0`, let `to` be `max((len + relativeTarget), 0)`; else let `to` be `min(relativeTarget, len)`.
5. Let `relativeStart` be `ToInteger(start)`.
6. If `relativeStart < 0`, let `from` be `max((len + relativeStart), 0)`; else let `from` be `min(relativeStart, len)`.
7. If `end` is `undefined`, let `relativeEnd` be `len`; else let `relativeEnd` be `ToInteger(end)`.
8. If `relativeEnd < 0`, let `final` be `max((len + relativeEnd), 0)`; else let `final` be `min(relativeEnd, len)`.
9. Let `count` be `min(final-from, len-to)`.
10. If `from<to` and `to<from+count`, then
    a. Let `direction` be `-1`.
    b. Let `from` be `from + count - 1`.
    c. Let `to` be `to + count - 1`.
11. Else,
    a. Let `direction` be `1`.
12. Repeat, while `count > 0`
    a. Let `fromKey` be `! ToString(from)`.
    b. Let `toKey` be `! ToString(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 `fromPresent` is `false`,
        i. Perform `DeletePropertyOrThrow(O, toKey)`.
    f. Let `from` be `from + direction`.
    g. Let `to` be `to + direction`.
    h. Let `count` be `count - 1`.
13. Return `O`. 
The `copyWithin` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

### 22.1.3.4 Array.prototype.entries ( )

The following steps are taken:

1. Let `O` be `? ToObject(this value)`.
2. Return `CreateArrayIterator(O, "key+value")`.

### 22.1.3.5 Array.prototype.every ( `callbackfn` [, `thisArg` ] )

`callbackfn` should be a function that accepts three arguments and returns a value that is coercible to the Boolean value `true` or `false`. `every` calls `callbackfn` once for each element present in the array, in ascending order, until it finds one where `callbackfn` returns `false`. If such an element is found, `every` immediately returns `false`. Otherwise, if `callbackfn` returned `true` for all elements, `every` will return `true`. `callbackfn` is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of `callbackfn`. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`every` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `every` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `every` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `callbackfn` will be the value at the time `every` visits them; elements that are deleted after the call to `every` begins and before being visited are not visited. `every` acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns `true`.

When the `every` method is called with one or two arguments, the following steps are taken:

1. Let `O` be `? ToObject(this value)`.
2. Let `len` be `? ToLength(? Get(O, "length"))`.
3. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
4. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
5. Let `k` be `0`. 
6. Repeat, while $k < len$
   a. Let $Pk$ be !`ToString(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, T, « kValue, k, O »))`.
      iii. If $testResult$ is `false`, return `false`.
   d. Increase $k$ by 1.
7. 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.

### 22.1.3.6 Array.prototype.fill ( value [, start [, end ] ] )

The `fill` method takes up to three arguments `value`, `start` and `end`.

**NOTE 1** The `start` and `end` arguments are optional with default values of 0 and the length of the `this` object. If `start` is negative, it is treated as `length+start` where `length` is the length of the array. If `end` is negative, it is treated as `length+end`.

The following steps are taken:

1. Let $O$ be ?`ToObject(this value)`.
2. Let $len$ be ?`ToLength(? Get(O, "length"))`.
3. Let $relativeStart$ be ?`ToInteger(start)`.
4. If $relativeStart < 0$, let $k$ be `max((len + relativeStart), 0)`; else let $k$ be `min(relativeStart, len)`.
5. If `end` is `undefined`, let $relativeEnd$ be `len`; else let $relativeEnd$ be ?`ToInteger(end)`.
6. If $relativeEnd < 0$, let $final$ be `max((len + relativeEnd), 0)`; else let $final$ be `min(relativeEnd, len)`.
7. Repeat, while $k < final$
   a. Let $Pk$ be !`ToString(k)`.
   b. Perform ?`Set(O, Pk, value, true)`.
   c. Increase $k$ by 1.
8. Return $O$.

**NOTE 2** The `fill` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

### 22.1.3.7 Array.prototype.filter ( callbackfn [, thisArg ] )

**NOTE 1** `callbackfn` should be a function that accepts three arguments and returns a value that is coercible to the Boolean value `true` or `false`. `filter` calls `callbackfn` once for each element
in the array, in ascending order, and constructs a new array of all the values for which `callbackfn` returns true. `callbackfn` is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a `thisArg` parameter is provided, it will be used as the this value for each invocation of `callbackfn`. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`filter` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `filter` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `filter` begins will not be visited by `callbackfn`. If existing elements of the array are changed their value as passed to `callbackfn` will be the value at the time `filter` visits them; elements that are deleted after the call to `filter` begins and before being visited are not visited.

When the `filter` method is called with one or two arguments, the following steps are taken:

1. Let `O` be `ToObject(this value)`.
2. Let `len` be `ToLength(? Get(O, "length"))`.
3. If `IsCallable(callbackfn)` is false, throw a `TypeError` exception.
4. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
5. Let `A` be `ArraySpeciesCreate(O, 0)`.
6. Let `k` be 0.
7. Let `to` be 0.
8. Repeat, while `k < len`
   a. Let `Pk` be `ToString(k)`.
   b. Let `kPresent` be `HasProperty(O, Pk)`.
   c. If `kPresent` is true, then
      i. Let `kValue` be `Get(O, Pk)`.
      ii. Let `selected` be `ToBoolean(? Call(callbackfn, T, « kValue, k, O »))`.
      iii. If `selected` is true, then
          1. Perform `CreateDataPropertyOrThrow(A, ! ToString(to), kValue)`.
          2. Increase `to` by 1.
   d. Increase `k` by 1.

**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.
22.1.3.8 **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 **callbackfn**. Elements that are appended to the array after the call to **find** begins will not be visited by **callbackfn**. If existing elements of the array are changed, their value as passed to *predicate* will be the value at the time that **find** visits them.

When the **find** method is called, the following steps are taken:

1. Let *O* be ? **ToObject**(this value).
2. Let *len* be ? **ToLength**(Get(*O*, "length")).
3. If **IsCallable**(predicate) is **false**, throw a **TypeError** exception.
4. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
5. Let *k* be 0.
6. Repeat, while *k* < *len*
   a. Let *Pk* be ! **ToString**(k).
   b. Let *kValue* be ? **Get**(0, *Pk*).
   c. Let *testResult* be **ToBoolean**(Call(*predicate*, *T*, « *kValue*, *k*, *O* »)).
   d. If *testResult* is **true**, return *kValue*.
   e. Increase *k* by 1.
7. 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.

22.1.3.9 **Array.prototype.findIndex** (*predicate*, *thisArg*)

**NOTE 1**

*predicate* should be a function that accepts three arguments and returns a value that is
coercible to the Boolean value true or false. `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 `callbackfn`. Elements that are appended to the array after the call to `findIndex` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `predicate` will be the value at the time that `findIndex` visits them.

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 `ToLength(? `Get(O, "length"))`.
3. If `IsCallable(predicate)` is false, throw a `TypeError` exception.
4. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
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(predicate, T, « kValue, k, O »))`.
   d. If `testResult` is true, return `k`.
   e. Increase `k` by 1.
7. Return -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.

### 22.1.3.10 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`.

When the `forEach` method is called with one or two arguments, the following steps are taken:

1. Let `O` be `? ToObject(this value)`.
2. Let `len` be `? ToLength(? Get(O, "length"))`.
3. If `isCallable(callbackfn)` is `false`, throw a `TypeError` exception.
4. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
5. Let `k` be `0`.
6. Repeat, while `k < len`
   a. Let `Pk` be `? ToString(k)`.
   b. Let `kPresent` be `? HasProperty(O, Pk)`.
   c. If `kPresent` is `true`, then
      i. Let `kValue` be `? Get(O, Pk)`.
      ii. Perform `? Call(callbackfn, T, « kValue, k, O »)`.
   d. Increase `k` by `1`.
7. 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.

### 22.1.3.11 Array.prototype.includes ( `searchElement` [ , `fromIndex` ] )

**NOTE 1**
`includes` compares `searchElement` to the elements of the array, in ascending order, using the `SameValueZero` algorithm, and if found at any position, returns `true`; otherwise, `false` is returned.

The optional second argument `fromIndex` defaults to `0` (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 negative, it is used as the offset from the end of the array to compute `fromIndex`. If the computed index is less than `0`, the whole array will be searched.

When the `includes` method is called, the following steps are taken:

1. Let `O` be `? ToObject(this value)`.
2. Let `len` be `? ToLength(? Get(O, "length"))`. 
3. If \( len \) is 0, return false.

4. Let \( n \) be \( \text{ToInteger(fromIndex)} \). (If \( \text{fromIndex} \) is undefined, this step produces the value 0.)

5. If \( n \geq 0 \), then
   a. Let \( k \) be \( n \).

6. Else \( n < 0 \),
   a. Let \( k \) be \( \text{len} + n \).
   b. If \( k < 0 \), let \( k \) be 0.

7. Repeat, while \( k < \text{len} \)
   a. Let \( \text{elementK} \) be the result of \( \text{Get(O, !ToString(k))} \).
   b. If \( \text{SameValueZero(searchElement, elementK)} \) is true, return true.
   c. Increase \( k \) by 1.

8. Return false.

**NOTE 2** The includes 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.

**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.

22.1.3.12 Array.prototype.indexOf (searchElement [, fromIndex ])

**NOTE 1** indexOf compares searchElement to the elements of the array, in ascending order, using the Strict Equality Comparison algorithm, and if found at one or more indices, returns the smallest such index; otherwise, -1 is returned.

The optional second argument fromIndex defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute fromIndex. If the computed index is less than 0, the whole array will be searched.

When the indexOf method is called with one or two arguments, the following steps are taken:

1. Let \( O \) be \( \text{ToObject(this value)} \).
2. Let \( \text{len} \) be \( \text{ToLength(? Get(O, "length"))} \).
3. If \( \text{len} \) is 0, return -1.
4. Let \( n \) be \( \text{ToInteger(fromIndex)} \). (If fromIndex is undefined, this step produces the value 0.)
5. If \( n \geq \text{len} \), return -1.
6. If \( n \geq 0 \), then
   a. If \( n \) is -0, let \( k \) be +0; else let \( k \) be \( n \).
6. Else \( n < 0 \),

Let $k$ be $len + n$.

b. If $k < 0$, let $k$ be 0.

8. 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. Increase $k$ by 1.


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.

22.1.3.13 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 ? ToLength(? Get($O$, "length")).
3. If separator is undefined, let sep be the single-element String ",, ".
4. Else, let sep be ? ToString(separator).
5. Let $R$ be the empty String.
6. Let $k$ be 0.
7. Repeat, while $k < len$
   a. If $k > 0$, let $R$ be a String value produced by concatenating $R$ and sep.
   b. Let $element$ be ? Get($O$, !ToString($k$)).
   c. If $element$ is undefined or null, let $next$ be the empty String; otherwise, let $next$ be ? ToString($element$).
   d. Set $R$ to a String value produced by concatenating $R$ and $next$.
   e. Increase $k$ by 1.
8. Return $R$.

NOTE 2  The join function is intentionally generic; it does not require that its this value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.

22.1.3.14 Array.prototype.keys ()
The following steps are taken:

1. Let \( O \) be ? \texttt{ToObject}(this value).
2. Return \texttt{CreateArrayIterator}(O, "key").

22.1.3.15 \texttt{Array.prototype.lastIndexOf} (searchElement [, fromIndex ])

**NOTE 1** \texttt{lastIndexOf} compares searchElement to the elements of the array in descending order using the \textit{Strict Equality Comparison} algorithm, and if found at one or more indices, returns the largest such index; otherwise, -1 is returned.

The optional second argument \texttt{fromIndex} defaults to the array's length minus one (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute \texttt{fromIndex}. If the computed index is less than 0, -1 is returned.

When the \texttt{lastIndexOf} method is called with one or two arguments, the following steps are taken:

1. Let \( O \) be ? \texttt{ToObject}(this value).
2. Let \( \text{len} \) be ? \texttt{ToLength}(? \texttt{Get}(O, "length")).
3. If \( \text{len} \) is 0, return -1.
4. If argument \texttt{fromIndex} was passed, let \( n \) be ? \texttt{ToInteger}(\texttt{fromIndex}); else let \( n \) be \text{len}-1.
5. If \( n \geq 0 \), then
   a. If \( n \) is -0, let \( k \) be +0; else let \( k \) be \text{min}(n, \text{len}-1).
6. Else \( n < 0 \),
   a. Let \( k \) be \text{len} + n.
7. Repeat, while \( k \geq 0 \)
   a. Let \( kPresent \) be ? \texttt{HasProperty}(O, ! \texttt{ToString}(k)).
   b. If \( kPresent \) is \text{true}, then
      i. Let \( elementK \) be ? \texttt{Get}(O, ! \texttt{ToString}(k)).
      ii. Let \( same \) be the result of performing \textit{Strict Equality Comparison} searchElement \( === \) elementK.
      iii. If \( same \) is \text{true}, return \( k \).
   c. Decrease \( k \) by 1.
8. Return -1.

**NOTE 2** The \texttt{lastIndexOf} function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.16 \texttt{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 ? `ToLength`(? `Get`(O, "length")).
3. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
4. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
5. Let `A` be ? `ArraySpeciesCreate`(O, len).
6. Let `k` be 0.
7. Repeat, while `k < len`
   a. Let `Pk` be ! `ToString`(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, T, « `kValue`, k, O »).
      iii. Perform ? `CreateDataPropertyOrThrow`(A, Pk, mappedValue).
   d. Increase `k` by 1.
8. Return `A`.

NOTE 2  

The `map` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.17  `Array.prototype.pop()`
When the `pop` method is called, the following steps are taken:

1. Let `O` be `ToObject(this value)`.
2. Let `len` be `ToLength(? Get(O, "length"))`.
3. If `len` is zero, then
   a. Perform `Set(O, "length", 0, true)`.
   b. Return `undefined`.
4. Else `len > 0`,
   a. Let `newLen` be `len-1`.
   b. Let `index` be `! ToString(newLen)`.
   c. Let `element` be `? Get(O, index)`.
   d. Perform `DeletePropertyOrThrow(O, index)`.
   e. Perform `Set(O, "length", newLen, true)`.
   f. Return `element`.

The `pop` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

**22.1.3.18 Array.prototype.push ( ...items )**

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 `ToLength(? Get(O, "length"))`.
3. Let `items` be a `List` whose elements are, in left to right order, the arguments that were passed to this function invocation.
4. Let `argCount` be the number of elements in `items`.
5. If `len + argCount > 2^{53}-1`, throw a `TypeError` exception.
6. Repeat, while `items` is not empty
   a. Remove the first element from `items` and let `E` be the value of the element.
   b. Perform `Set(O, ! ToString(len), E, true)`.
   c. Let `len` be `len+1`.
7. Perform `Set(O, "length", len, true)`.
8. Return `len`.

The `length` property of the `push` method is 1.

The `push` function is intentionally generic; it does not require that its `this` value be an
Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.19 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 provided in the call to reduce, then previousValue will be equal to initialValue and currentValue will be equal to the first value in the array. If no initialValue was provided, then previousValue will be equal to the first value in the array and currentValue will be equal to the second. It is a TypeError if the array contains no elements and initialValue is not provided.

reduce does not directly mutate the object on which it is called but the object may be mutated by the calls to callbackfn.

The range of elements processed by reduce is set before the first call to callbackfn. Elements that are appended to the array after the call to reduce begins will not be visited by callbackfn. If existing elements of the array are changed, their value as passed to callbackfn will be the value at the time reduce visits them; elements that are deleted after the call to reduce begins and before being visited are not visited.

When the reduce method is called with one or two arguments, the following steps are taken:

1. Let O be ? ToObject(this value).
2. Let len be ? ToLength(? Get(O, "length")).
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 0.
6. If initialValue is present, then
   a. Set accumulator to initialValue.
7. Else initialValue is not present,
   a. Let kPresent be false.
   b. Repeat, while kPresent is false and k < len
      i. Let Pk be ! ToString(k).
      ii. Let kPresent be ? HasProperty(O, Pk).
      iii. If kPresent is true, then
         1. Let accumulator be ? Get(O, Pk).
iv. Increase \( k \) by 1.

\[
\text{c. If } k\text{Present is } \text{false}, \text{ throw a } \text{TypeError} \text{ exception.}
\]

8. Repeat, while \( k < \text{len} \)

a. Let \( Pk \) be \( ! \text{ToString}(k) \).

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 \( \text{accumulator} \) be \( ? \text{Call}(\text{callbackfn}, \text{undefined}, \text{«} \text{accumulator}, k\text{Value}, k, O \text{»}) \).

d. Increase \( k \) by 1.

9. Return \( \text{accumulator} \).

NOTE 2 The \text{reduce} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.20 \text{Array.prototype.reduceRight} ( \text{callbackfn} [, \text{initialValue} ] )

NOTE 1 \text{callbackfn} should be a function that takes four arguments. \text{reduceRight} calls the callback, as a function, once for each element after the first element present in the array, in descending order.

\text{callbackfn} is called with four arguments: the \text{previousValue} (value from the previous call to \text{callbackfn}), the \text{currentValue} (value of the current element), the \text{currentIndex}, and the object being traversed. The first time the function is called, the \text{previousValue} and \text{currentValue} can be one of two values. If an \text{initialValue} was provided in the call to \text{reduceRight}, then \text{previousValue} will be equal to \text{initialValue} and \text{currentValue} will be equal to the last value in the array. If no \text{initialValue} was provided, then \text{previousValue} will be equal to the last value in the array and \text{currentValue} will be equal to the second-to-last value. It is a \text{TypeError} if the array contains no elements and \text{initialValue} is not provided.

\text{reduceRight} 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{reduceRight} is set before the first call to \text{callbackfn}. Elements that are appended to the array after the call to \text{reduceRight} begins will not be visited by \text{callbackfn}. If existing elements of the array are changed by \text{callbackfn}, their value as passed to \text{callbackfn} will be the value at the time \text{reduceRight} visits them; elements that are deleted after the call to \text{reduceRight} begins and before being visited are not visited.

When the \text{reduceRight} 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 `ToLength(? Get(O, "length"))`.
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. If `initialValue` is present, then
   a. Set `accumulator` to `initialValue`.
7. Else `initialValue` is not present,
   a. Let `kPresent` be `false`.
   b. Repeat, while `kPresent` is `false` and `k ≥ 0`
      i. Let `Pk` be `ToString(k)`.
      ii. Let `kPresent` be `HasProperty(O, Pk)`.
      iii. If `kPresent` is `true`, then
         1. Let `accumulator` be `Get(O, Pk)`.
      iv. Decrease `k` by 1.
   c. If `kPresent` is `false`, throw a `TypeError` exception.
8. Repeat, while `k ≥ 0`
   a. Let `Pk` be `ToString(k)`.
   b. Let `kPresent` be `HasProperty(O, Pk)`.
   c. If `kPresent` is `true`, then
      i. Let `kValue` be `Get(O, Pk)`.
      ii. Let `accumulator` be `Call(callbackfn, undefined, « accumulator, kValue, k, O »)`.
   d. Decrease `k` by 1.

**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.

**22.1.3.21** 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 `ToLength(? Get(O, "length"))`.
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(upper).
c. Let lowerP be !ToString(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 both lowerExists and upperExists are false,
   i. No action is required.
   l. Increase lower by 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.

22.1.3.22 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 ? ToLength(? Get(O, "length")).
3. If len is zero, 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(k).
   b. Let to be ! ToString(k-1).
The following steps are taken:

1. Let \( O \) be ? ToObject(this value).
2. Let \( len \) be ? ToLength(? Get(O, "length").
3. Let relativeStart be ? ToInteger(start).
4. If relativeStart < 0, let \( k \) be \( \max((len + relativeStart), 0) \); else let \( k \) be \( \min(relativeStart, len) \).
5. If end is undefined, let relativeEnd be len; else let relativeEnd be ? ToInteger(end).
6. If relativeEnd < 0, let final be \( \max((len + relativeEnd), 0) \); else let final be \( \min(relativeEnd, len) \).
7. Let count be \( \max(final - k, 0) \).
8. Let \( A \) be ? ArraySpeciesCreate(O, count).
9. Let \( n \) be 0.
10. Repeat, while \( k < final \)
    a. Let \( Pk \) be ! ToString(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(n), kValue).
    d. Increase \( k \) by 1.
    e. Increase \( n \) by 1.
12. Return A.

NOTE 2  The explicit setting of the length property of the result Array in step 11 was necessary in previous editions of ECMAScript to ensure that its length was correct in situations where the trailing elements of the result Array were not present. Setting length became unnecessary starting in ES2015 when the result Array was initialized to its proper length rather than an empty Array but is carried forward to preserve backward compatibility.

NOTE 3  The slice function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.24  Array.prototype.some ( callbackfn [, thisArg ] )

NOTE 1  callbackfn should be a function that accepts three arguments and returns a value that is coercible to the Boolean value true or false. some calls callbackfn once for each element present in the array, in ascending order, until it finds one where callbackfn returns true. If such an element is found, some immediately returns true. Otherwise, some returns false. callbackfn is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a thisArg parameter is provided, it will be used as the this value for each invocation of callbackfn. If it is not provided, undefined is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

some does not directly mutate the object on which it is called but the object may be mutated by the calls to callbackfn.

The range of elements processed by some is set before the first call to callbackfn. Elements that are appended to the array after the call to some begins will not be visited by callbackfn. If existing elements of the array are changed, their value as passed to callbackfn will be the value at the time that some visits them; elements that are deleted after the call to some begins and before being visited are not visited. some acts like the "exists" quantifier in mathematics. In particular, for an empty array, it returns false.

When the some method is called with one or two arguments, the following steps are taken:

1. Let O be ? ToObject(this value).
2. Let len be ? ToLength(? Get(O, "length")).
3. If IsCallable(callbackfn) is false, throw a TypeError exception.
4. If thisArg was supplied, let T be thisArg; else let T be undefined.
5. Let $k$ be 0.
6. Repeat, while $k < len$
   a. Let $Pk$ be $\text{ToString}(k)$.
   b. Let $k\text{Present}$ be $\text{? HasProperty}(O, Pk)$.
   c. If $k\text{Present}$ is true, then
      i. Let $kValue$ be $\text{Get}(O, Pk)$.
      ii. Let $test\text{Result}$ be $\text{ToBoolean}(\text{? Call}(callbackfn, T, \langle kValue, k, O \rangle))$.
      iii. If $test\text{Result}$ is true, return true.
   d. Increase $k$ by 1.
7. Return false.

**NOTE 2**  
The `some` function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

### 22.1.3.25 Array.prototype.sort (comparefn)

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If comparefn is not `undefined`, it should be a function that accepts two arguments $x$ and $y$ and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

Upon entry, the following steps are performed to initialize evaluation of the sort function:

1. Let $obj$ be $\text{ToObject}(\text{this value})$.
2. Let $len$ be $\text{ToLength}(\text{? Get}(obj, \"length\") )$.

Within this specification of the sort method, an object, $obj$, is said to be sparse if the following algorithm returns true:

1. For each integer $i$ in the range $0 \leq i < len$, do
   a. Let $elem$ be $obj.\text{[[GetOwnProperty]]}(\text{! ToString}(i))$.
   b. If $elem$ is `undefined`, return true.
2. Return false.

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:

If comparefn is not `undefined` and is not a consistent comparison function for the elements of this array (see below), the sort order is implementation-defined. The sort order is also implementation-defined if comparefn is `undefined` and SortCompare does not act as a consistent comparison function.

Let $proto$ be $obj.\text{[[GetPrototypeOf]]}()$. If $proto$ is not `null` and there exists an integer $j$ such that all of the conditions below are satisfied then the sort order is implementation-defined:
• **obj** is sparse
• \(0 \leq j < \text{len}\)
• \(\text{HasProperty}(\text{proto}, \text{ToString}(j))\) is true.

The sort order is also implementation-defined if **obj** is sparse and any of the following conditions are true:

• \(\text{IsExtensible}(\text{obj})\) is false.
• Any integer index property of **obj** whose name is a nonnegative integer less than \(\text{len}\) is a data property whose [[Configurable]] attribute is false.

The sort order is also implementation-defined if any of the following conditions are true:

• If **obj** is an exotic object (including Proxy exotic objects) whose behaviour for [[Get]], [[Set]], [[Delete]], and [[GetOwnProperty]] is not the ordinary object implementation of these internal methods.
• If any index property of **obj** whose name is a nonnegative integer less than \(\text{len}\) is an accessor property or is a data property whose [[Writable]] attribute is false.
• If \(\text{comparefn}\) is undefined and the application of ToString to any value passed as an argument to SortCompare modifies **obj** or any object on **obj**'s prototype chain.
• If \(\text{comparefn}\) is undefined and all applications of ToString, to any specific value passed as an argument to SortCompare, do not produce the same result.

The following steps are taken:

1. Perform an implementation-dependent sequence of calls to the [[Get]] and [[Set]] internal methods of **obj**, to the DeletePropertyOrThrow and HasOwnProperty abstract operation with **obj** as the first argument, and to SortCompare (described below), such that:
   • The property key argument for each call to [[Get]], [[Set]], HasOwnProperty, or DeletePropertyOrThrow is the string representation of a nonnegative integer less than \(\text{len}\).
   • The arguments for calls to SortCompare are values returned by a previous call to the [[Get]] internal method, unless the properties accessed by those previous calls did not exist according to HasOwnProperty. If both perspective arguments to SortCompare correspond to non-existent properties, use +0 instead of calling SortCompare. If only the first perspective argument is non-existent use +1. If only the second perspective argument is non-existent use -1.
   • If **obj** is not sparse then DeletePropertyOrThrow must not be called.
   • If any [[Set]] call returns false a TypeError exception is thrown.
   • If an abrupt completion is returned from any of these operations, it is immediately returned as the value of this function.
2. Return **obj**.

Unless the sort order is specified above to be implementation-defined, the returned object must have the following two characteristics:
• There must be some mathematical permutation \( \pi \) of the nonnegative integers less than \( \text{len} \), such that for every nonnegative integer \( j \) less than \( \text{len} \), if property \( \text{old}[j] \) existed, then \( \text{new}[\pi(j)] \) is exactly the same value as \( \text{old}[j] \). But if property \( \text{old}[j] \) did not exist, then \( \text{new}[\pi(j)] \) does not exist.
• Then for all nonnegative integers \( j \) and \( k \), each less than \( \text{len} \), if \( \text{SortCompare}\,(\text{old}[j], \text{old}[k]) < 0 \) (see \( \text{SortCompare} \) below), then \( \text{new}[\pi(j)] < \text{new}[\pi(k)] \).

Here the notation \( \text{old}[j] \) is used to refer to the hypothetical result of calling \( \text{obj}.\,[\text{Get}](j) \) before this function is executed, and the notation \( \text{new}[j] \) to refer to the hypothetical result of calling \( \text{obj}.\,[\text{Get}](j) \) after this function has been executed.

A function \( \text{comparefn} \) is a consistent comparison function for a set of values \( S \) if all of the requirements below are met for all values \( a, b, \) and \( c \) (possibly the same value) in the set \( S \): The notation \( a <_{\text{CF}} b \) means \( \text{comparefn}(a, b) < 0 \); \( a =_{\text{CF}} b \) means \( \text{comparefn}(a, b) = 0 \) (of either sign); and \( a >_{\text{CF}} b \) means \( \text{comparefn}(a, b) > 0 \).

• Calling \( \text{comparefn}(a, b) \) always returns the same value \( v \) when given a specific pair of values \( a \) and \( b \) as its two arguments. Furthermore, \( \text{Type}(v) \) is \text{Number}, and \( v \) is not \( \text{NaN} \). Note that this implies that exactly one of \( a <_{\text{CF}} b, a =_{\text{CF}} b, \) and \( a >_{\text{CF}} b \) will be true for a given pair of \( a \) and \( b \).
• Calling \( \text{comparefn}(a, b) \) does not modify \( \text{obj} \) or any object on \( \text{obj} \)'s prototype chain.
  \( a =_{\text{CF}} a \) (reflexivity)
  \( \text{If } a =_{\text{CF}} b, \text{ then } b =_{\text{CF}} a \) (symmetry)
  \( \text{If } a =_{\text{CF}} b \text{ and } b =_{\text{CF}} c, \text{ then } a =_{\text{CF}} c \) (transitivity of \( =_{\text{CF}} \))
  \( \text{If } a <_{\text{CF}} b \text{ and } b <_{\text{CF}} c, \text{ then } a <_{\text{CF}} c \) (transitivity of \( <_{\text{CF}} \))
  \( \text{If } a >_{\text{CF}} b \text{ and } b >_{\text{CF}} c, \text{ then } a >_{\text{CF}} c \) (transitivity of \( >_{\text{CF}} \))

**NOTE 1**  
The above conditions are necessary and sufficient to ensure that \( \text{comparefn} \) divides the set \( S \) into equivalence classes and that these equivalence classes are totally ordered.

**NOTE 2**  
The \text{sort} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 22.1.3.25.1 Runtime Semantics: \text{SortCompare}( \text{x, y} )

The \text{SortCompare} abstract operation is called with two arguments \( x \) and \( y \). It also has access to the \( \text{comparefn} \) argument passed to the current invocation of the \text{sort} method. The following steps are taken:

1. If \( x \) and \( y \) are both \text{undefined}, return \(+0\).
2. If \( x \) is \text{undefined}, return \(1\).
3. If \( y \) is \text{undefined}, return \(-1\).
4. If the argument \( \text{comparefn} \) is not \text{undefined}, then
   a. Let \( v \) be \( \text{? ToNumber}(\text{? Call}(\text{comparefn}, \text{undefined}, \{x, y\})) \).
   b. If \( v \) is \text{NaN}, return \(+0\).
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.
9. Let ySmaller be the result of performing Abstract Relational Comparison yString < xString.
10. If ySmaller is true, return 1.
11. Return +0.

NOTE 1 Because non-existent property values always compare greater than undefined property values, and undefined always compares greater than any other value, undefined property values always sort to the end of the result, followed by non-existent property values.

NOTE 2 Method calls performed by the ToString abstract operations in steps 5 and 7 have the potential to cause SortCompare to not behave as a consistent comparison function.

22.1.3.26 Array.prototype.splice ( start, deleteCount, ...items )

NOTE 1 When the splice method is called with two or more arguments start, deleteCount and zero or more items, the deleteCount elements of the array starting at integer index start are replaced by the arguments 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 ? ToLength(? Get(O, "length")).
3. Let relativeStart be ? ToInteger(start).
4. If relativeStart < 0, let actualStart be max((len + relativeStart), 0); else let actualStart be min(relativeStart, len).
5. If the number of actual arguments is 0, then
   a. Let insertCount be 0.
   b. Let actualDeleteCount be 0.
6. Else if the number of actual arguments is 1, then
   a. Let insertCount be 0.
   b. Let actualDeleteCount be len - actualStart.
7. Else,
   a. Let insertCount be the number of actual arguments minus 2.
   b. Let dc be ? ToInteger(deleteCount).
   c. Let actualDeleteCount be min(max(dc, 0), len - actualStart).
8. If len+insertCount-actualDeleteCount > 2^{53}-1, throw a TypeError exception.
10. Let \( k \) be 0.
11. Repeat, while \( k < \text{actualDeleteCount} \)
    a. Let from be ! ToString(actualStart+k).
    b. Let fromPresent be ? HasProperty(0, from).
    c. If fromPresent is true, then
       i. Let fromValue be ? Get(0, from).
       ii. Perform ? CreateDataPropertyOrThrow(A, ! ToString(k), fromValue).
    d. Increment \( k \) by 1.
13. Let items be a List whose elements are, in left to right order, the portion of the actual argument list starting with the third argument. The list is empty if fewer than three arguments were passed.
14. Let itemCount be the number of elements in items.
15. If itemCount < actualDeleteCount, then
    a. Let \( k \) be actualStart.
    b. Repeat, while \( k < (len - \text{actualDeleteCount}) \)
       i. Let from be ! ToString(k+actualDeleteCount).
       ii. Let to be ! ToString(k+itemCount).
       iii. Let fromPresent be ? HasProperty(0, from).
       iv. If fromPresent is true, then
           1. Let fromValue be ? Get(0, from).
           2. Perform ? Set(0, to, fromValue, true).
       v. Else fromPresent is false,
           1. Perform ? DeletePropertyOrThrow(0, to).
    c. Let \( k \) be len.
    d. Repeat, while \( k > (len - \text{actualDeleteCount} + \text{itemCount}) \)
       i. Perform ? DeletePropertyOrThrow(0, ! ToString(k-1)).
       ii. Decrease \( k \) by 1.
16. Else if itemCount > actualDeleteCount, then
    a. Let \( k \) be \((len - \text{actualDeleteCount})\).
    b. Repeat, while \( k > \text{actualStart} \)
       i. Let from be ! ToString\((k + \text{actualDeleteCount} - 1)\).
       ii. Let to be ! ToString\((k + \text{itemCount} - 1)\).
       iii. Let fromPresent be ? HasProperty(0, from).
       iv. If fromPresent is true, then
           1. Let fromValue be ? Get(0, from).
           2. Perform ? Set(0, to, fromValue, true).
       v. Else fromPresent is false,
           1. Perform ? DeletePropertyOrThrow(0, to).
    vi. Decrease \( k \) by 1.
17. Let \( k \) be \( \text{actualStart} \).

18. Repeat, while \( \text{items} \) is not empty
   a. Remove the first element from \( \text{items} \) and let \( E \) be the value of that element.
   b. Perform \( \text{Set}(O, ! \text{ToString}(k), E, \text{true}) \).
   c. Increase \( k \) by 1.

19. Perform \( \text{Set}(O, \text{"length"}, \text{len} - \text{actualDeleteCount} + \text{itemCount}, \text{true}) \).

20. Return \( A \).

NOTE 2 The explicit setting of the \text{length} property of the result Array in step 19 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 \text{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 \text{splice} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

### 22.1.3.27 Array.prototype.toLocaleString ([[reserved1] [, reserved2]])

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the \text{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 \text{toLocaleString} method is used.

NOTE 1 The first edition of ECMA-402 did not include a replacement specification for the \text{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 \text{array} be \( \text{ToObject(this value)} \).
2. Let \text{len} be \( \text{ToLength(? Get(array, "length"))} \).
3. Let \text{separator} be the String value for the list-separator String appropriate for the host environment’s current locale (this is derived in an implementation-defined way).
4. Let \( R \) be the empty String.
5. Let \( k \) be 0.
6. Repeat, while \( k < \text{len} \)
   a. If \( k > 0 \), then
      i. Set \( R \) to a String value produced by concatenating \( R \) and \text{separator}.
b. Let `nextElement` be `Get(array, !ToString(k))`.
c. If `nextElement` is not `undefined` or `null`, then
   i. Let `S` be `ToString(? Invoke(nextElement, "toLocaleString"))`.
   ii. Set `R` to a String value produced by concatenating `R` and `S`.
   
   d. Increase `k` by 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.

### 22.1.3.28 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`, let `func` be 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.

### 22.1.3.29 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 `ToLength(? Get(O, "length"))`.
3. Let `argCount` be the number of actual arguments.
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 \( fromPresent \) is \( false \),
      1. Perform \( ? \) \( DeletePropertyOrThrow(O, to) \).
   vi. Decrease \( k \) by 1.

d. Let \( j \) be 0.

e. Let \( items \) be a \( List \) whose elements are, in left to right order, the arguments that were passed to this function invocation.

f. Repeat, while \( items \) is not empty
   i. Remove the first element from \( items \) and let \( E \) be the value of that element.
   ii. Perform \( ? \) \( Set(O, ! \) \( ToString(j), E, true) \).
   iii. Increase \( j \) by 1.
   5. Perform \( ? \) \( Set(O, "length", len+argCount, true) \).
   6. Return \( len+argCount \).

The \( length \) property of the \unshift\ method is 1.

\begin{enumerate}
\item The \unshift function is intentionally generic; it does not require that its \texttt{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.
\end{enumerate}

\subsection{22.1.30 \array.prototype.values ( )}

The following steps are taken:

1. Let \( O \) be \( ? \) \( ToObject(this value) \).
2. Return \( CreateArrayIterator(O, "value") \).

This function is the \%ArrayProto_values% intrinsic object.

\subsection{22.1.31 \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.

\subsection{22.1.32 \array.prototype [ @@unscopables ]
The initial value of the @@unscopables data property is an object created by the following steps:

1. Let `unscopableList` be `ObjectCreate(null)`.
2. Perform `CreateDataProperty(unscopableList, "copyWithin", true)`.
3. Perform `CreateDataProperty(unscopableList, "entries", true)`.
4. Perform `CreateDataProperty(unscopableList, "fill", true)`.
5. Perform `CreateDataProperty(unscopableList, "find", true)`.
6. Perform `CreateDataProperty(unscopableList, "findIndex", true)`.
7. Perform `CreateDataProperty(unscopableList, "includes", true)`.
8. Perform `CreateDataProperty(unscopableList, "keys", true)`.
9. Perform `CreateDataProperty(unscopableList, "values", true)`.
10. Assert: Each of the above calls will return `true`.
11. 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.

22.1.4 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.

22.1.4.1 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 9.4.2.1.

22.1.5 Array Iterator Objects

An Array Iterator is an object, that represents a specific iteration over some specific Array instance object. There is not a named constructor for Array Iterator objects. Instead, Array iterator objects are created by calling certain methods of Array instance objects.

22.1.5.1 CreateArrayIterator (array, kind)

Several methods of Array objects return Iterator objects. The abstract operation CreateArrayIterator with arguments array and kind is used to create such iterator objects. It performs the following steps:

1. Assert: Type(array) is Object.
2. Let iterator be ObjectCreate(%ArrayIteratorPrototype%, « [[IteratedObject]],
   [[ArrayIteratorNextIndex]], [[ArrayIterationKind]] »).
3. Set iterator.[[IteratedObject]] to array.
4. Set iterator.[[ArrayIteratorNextIndex]] to 0.
5. Set iterator.[[ArrayIterationKind]] to kind.
6. Return iterator.

22.1.5.2 The %ArrayIteratorPrototype% Object

All Array Iterator Objects inherit properties from the %ArrayIteratorPrototype% intrinsic object. The %ArrayIteratorPrototype% object is an ordinary object and its [[Prototype]] internal slot is the %IteratorPrototype% intrinsic object. In addition, %ArrayIteratorPrototype% has the following properties:

22.1.5.2.1 %ArrayIteratorPrototype%.next()

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have all of the internal slots of an Array Iterator Instance (22.1.5.3), throw a TypeError exception.
4. Let a be O.[[IteratedObject]].
5. If a is undefined, return CreateIterResultObject(undefined, true).
6. Let index be O.[[ArrayIteratorNextIndex]].
7. Let itemKind be O.[[ArrayIterationKind]].
8. If a has a [[TypedArrayName]] internal slot, then
   a. If IsDetachedBuffer(a.[[ViewedArrayBuffer]]) is true, throw a TypeError exception.
b. Let \( len \) be \( a.\text{[ArrayLength]} \).

9. Else,
   a. Let \( len \) be \( \text{ToLength}(? \text{Get}(a, "\text{length"}) \).

10. If \( index \geq len \), then
    a. Set \( a.\text{[IteratedObject]} \) to undefined.
    b. Return \( \text{CreateIterResultObject(undefined, true)} \).

11. Set \( a.\text{[ArrayIteratorNextIndex]} \) to \( index+1 \).

12. If \( \text{itemKind} \) is "key", return \( \text{CreateIterResultObject(index, false)} \).

13. Let \( \text{elementKey} \) be \( ! \text{ToString}(index) \).

14. Let \( \text{elementValue} \) be \(? \text{Get}(a, \text{elementKey}) \).

15. If \( \text{itemKind} \) is "value", let \( \text{result} \) be \( \text{elementValue} \).

16. Else,
   a. Assert: \( \text{itemKind} \) is "key+value".
   b. Let \( \text{result} \) be \( \text{CreateArrayFromList(«index, elementValue»)} \).

17. Return \( \text{CreateIterResultObject(result, false)} \).

### 22.1.5.2.2 %ArrayIteratorPrototype% [ @@toStringTag ]

The initial value of the \( @@\text{toStringTag} \) property is the String value "Array Iterator".

This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true \}.

### 22.1.5.3 Properties of Array Iterator Instances

Array Iterator instances are ordinary objects that inherit properties from the %ArrayIteratorPrototype% intrinsic object. Array Iterator instances are initially created with the internal slots listed in Table 49.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[IteratedObject]]</td>
<td>The object whose array elements are being iterated.</td>
</tr>
<tr>
<td>[[ArrayIteratorNextIndex]]</td>
<td>The integer index of the next integer index to be examined by this iteration.</td>
</tr>
<tr>
<td>[[ArrayIterationKind]]</td>
<td>A String value that identifies what is returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;.</td>
</tr>
</tbody>
</table>

### 22.2 TypedArray Objects

TypedArray objects present an array-like view of an underlying binary data buffer (24.1). Each element of a TypedArray instance has the same underlying binary scalar data type. There is a distinct TypedArray
constructor, listed in Table 50, for each of the nine supported element types. Each constructor in Table 50 has a corresponding distinct prototype object.

<table>
<thead>
<tr>
<th>Constructor Name and Intrinsic</th>
<th>Element Type</th>
<th>Element Size</th>
<th>Conversion Operation</th>
<th>Description</th>
<th>Equivalent C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Array %Int8Array%</td>
<td>Int8</td>
<td>1</td>
<td>ToInt8</td>
<td>8-bit 2’s complement signed integer</td>
<td>signed char</td>
</tr>
<tr>
<td>Uint8Array %Uint8Array%</td>
<td>Uint8</td>
<td>1</td>
<td>ToUint8</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
</tr>
<tr>
<td>Uint8ClampedArray %Uint8ClampedArray%</td>
<td>Uint8C</td>
<td>1</td>
<td>ToUint8Clamp</td>
<td>8-bit unsigned integer (clamped conversion)</td>
<td>unsigned char</td>
</tr>
<tr>
<td>Int16Array %Int16Array%</td>
<td>Int16</td>
<td>2</td>
<td>ToInt16</td>
<td>16-bit 2’s complement signed integer</td>
<td>short</td>
</tr>
<tr>
<td>Uint16Array %Uint16Array%</td>
<td>Uint16</td>
<td>2</td>
<td>ToUint16</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
</tr>
<tr>
<td>Int32Array %Int32Array%</td>
<td>Int32</td>
<td>4</td>
<td>ToInt32</td>
<td>32-bit 2’s complement signed integer</td>
<td>int</td>
</tr>
<tr>
<td>Uint32Array %Uint32Array%</td>
<td>Uint32</td>
<td>4</td>
<td>ToUint32</td>
<td>32-bit unsigned integer</td>
<td>unsigned int</td>
</tr>
<tr>
<td>Float32Array %Float32Array%</td>
<td>Float32</td>
<td>4</td>
<td></td>
<td>32-bit IEEE floating point</td>
<td>float</td>
</tr>
<tr>
<td>Float64Array %Float64Array%</td>
<td>Float64</td>
<td>8</td>
<td></td>
<td>64-bit IEEE floating point</td>
<td>double</td>
</tr>
</tbody>
</table>

In the definitions below, references to %TypedArray% should be replaced with the appropriate constructor name from the above table. The phrase “the element size in bytes” refers to the value in the Element Size column of the table in the row corresponding to the constructor. The phrase “element Type” refers to the value in the Element Type column for that row.

### 22.2.1 The %TypedArray% Intrinsic Object

The %TypedArray% intrinsic object is a constructor function object that all of the %TypedArray% constructor objects inherit from. %TypedArray% and its corresponding prototype object provide common properties
that are inherited by all \texttt{TypedArray} constructors and their instances. The \%TypedArray\% intrinsic does not have a global name or appear as a property of the \texttt{global object}.

The \%TypedArray\% intrinsic function object acts as the abstract superclass of the various \texttt{TypedArray} constructors. Because it is an abstract class constructor it will throw an error when invoked. The \texttt{TypedArray} constructors do not perform a super call to it.

### 22.2.1.1 \%TypedArray\% ( )

The \%TypedArray\% constructor performs the following steps:

1. Throw a \texttt{TypeError} exception.

The \texttt{length} property of the \%TypedArray\% constructor function is 0.

### 22.2.2 Properties of the \%TypedArray\% Intrinsic Object

The value of the \texttt{[[Prototype]]} internal slot of \%TypedArray\% is the intrinsic object \%FunctionPrototype\%.

The \texttt{name} property of the \%TypedArray\% constructor function is "\texttt{TypedArray}".

The \%TypedArray\% constructor has the following properties:

### 22.2.2.1 \%TypedArray\%.from ( source [ , mapfn [ , thisArg ] ] )

When the \texttt{from} method is called with argument \texttt{source}, and optional arguments \texttt{mapfn} and \texttt{thisArg}, the following steps are taken:

1. Let \texttt{C} be the \texttt{this} value.
2. If \texttt{IsConstructor(C)} is \texttt{false}, throw a \texttt{TypeError} exception.
3. If \texttt{mapfn} was supplied and \texttt{mapfn} is not \texttt{undefined}, then
   a. If \texttt{IsCallable(mapfn)} is \texttt{false}, throw a \texttt{TypeError} exception.
   b. Let \texttt{mapping} be \texttt{true}.
4. Else, let \texttt{mapping} be \texttt{false}.
5. If \texttt{thisArg} was supplied, let \texttt{T} be \texttt{thisArg}; else let \texttt{T} be \texttt{undefined}.
6. Let \texttt{usingIterator} be \texttt{? GetMethod(source, @@iterator)}.
7. If \texttt{usingIterator} is not \texttt{undefined}, then
   a. Let \texttt{values} be \texttt{? \texttt{IterableToList(source, usingIterator)}}.
   b. Let \texttt{len} be the number of elements in \texttt{values}.
   c. Let \texttt{targetObj} be \texttt{? \texttt{TypedArrayCreate(C, «len»)}}.
   d. Let \texttt{k} be 0.
   e. Repeat, while \texttt{k < len}
i. Let $P_k$ be $!\text{ToString}(k)$.
ii. Let $k\text{Value}$ be the first element of $values$ and remove that element from $values$.
iii. If $mapping$ is $\text{true}$, then
   1. Let $mappedValue$ be $?\text{Call}(\text{mapfn}, T, \langle k\text{Value}, k \rangle)$.
iv. Else, let $mappedValue$ be $k\text{Value}$.
   v. Perform $?\text{Set}(\text{targetObj}, P_k, mappedValue, \text{true})$.
vi. Increase $k$ by 1.

f. Assert: $values$ is now an empty List.
g. Return $\text{targetObj}$.

8. NOTE: $source$ is not an Iterable so assume it is already an array-like object.

9. Let $arrayLike$ be $!\text{ToObject}(source)$.
10. Let $len$ be $?\text{ToLength}(?\text{Get}(arrayLike, \text{"length"}))$.
11. Let $\text{targetObj}$ be $?\text{TypedArrayCreate}(C, \langle \text{len} \rangle)$.
12. Let $k$ be 0.
13. Repeat, while $k < len$
   a. Let $P_k$ be $!\text{ToString}(k)$.
   b. Let $k\text{Value}$ be $?\text{Get}(arrayLike, P_k)$.
   c. If $mapping$ is $\text{true}$, then
      i. Let $mappedValue$ be $?\text{Call}(\text{mapfn}, T, \langle k\text{Value}, k \rangle)$.
   d. Else, let $mappedValue$ be $k\text{Value}$.
   e. Perform $?\text{Set}(\text{targetObj}, P_k, mappedValue, \text{true})$.
   f. Increase $k$ by 1.
14. Return $\text{targetObj}$.

### 22.2.2.1.1 Runtime Semantics: IterableToList($items, method$)

The abstract operation $\text{IterableToList}$ performs the following steps:

1. Let $\text{iterator}$ be $?\text{GetIterator}(items, method)$.
2. Let $values$ be a new empty List.
3. Let $next$ be $\text{true}$.
4. Repeat, while $next$ is not $\text{false}$
   a. Set $next$ to $?\text{IteratorStep}(\text{iterator})$.
   b. If $next$ is not $\text{false}$, then
      i. Let $nextValue$ be $?\text{IteratorValue}(next)$.
      ii. Append $nextValue$ to the end of the List $values$.
5. Return $values$.

### 22.2.2 %TypedArray%.of (...$items$)

When the $\text{of}$ method is called with any number of arguments, the following steps are taken:
1. Let \( \text{len} \) be the actual number of arguments passed to this function.
2. Let \( \text{items} \) be the List of arguments passed to this function.
3. Let \( C \) be the \textbf{this} value.
4. If \( \text{IsConstructor}(C) \) is \textbf{false}, throw a \textbf{TypeError} exception.
5. Let \( \text{newObj} \) be ? \text{TypedArrayCreate}(C, \{ \text{len} \}).
6. Let \( k \) be \( 0 \).
7. Repeat, while \( k < \text{len} \)
   a. Let \( k\text{Value} \) be \( \text{items}[k] \).
   b. Let \( Pk \) be ! \text{ToString}(k).
   c. Perform ? \text{Set}(\text{newObj}, Pk, kValue, \text{true}).
   d. Increase \( k \) by \( 1 \).
8. Return \( \text{newObj} \).

\[\text{NOTE} \quad \text{The} \ \text{items} \ \text{argument is assumed to be a well-formed rest argument value.}\]

### 22.2.2.3 %TypedArray%.prototype

The initial value of %TypedArray%.prototype is the %TypedArrayPrototype% intrinsic object.

This property has the attributes \{ [[Writable]]: \textbf{false}, [[Enumerable]]: \textbf{false}, [[Configurable]]: \textbf{false} \}.

### 22.2.4 get %TypedArray% [ @@species ]

%TypedArray%[@@species] is an accessor property whose set accessor function is \textbf{undefined}. Its get accessor function performs the following steps:

1. Return the \textbf{this} value.

The value of the \textbf{name} property of this function is "get [Symbol.species]".

\[\text{NOTE} \quad \%\text{TypedArrayPrototype} \% \text{methods normally use their this \text{object's} constructor to create a derived object. However, a subclass constructor may over-ride that default behaviour by redefining its @@species property.} \]

### 22.2.3 Properties of the %TypedArrayPrototype% Object

The value of the [[Prototype]] internal slot of the %TypedArrayPrototype% object is the intrinsic object \%ObjectPrototype%. The %TypedArrayPrototype% object is an ordinary object. It does not have a [[ViewedArrayBuffer]] or any other of the internal slots that are specific to \textbf{TypedArray} instance objects.

### 22.2.3.1 get %TypedArray%.prototype.buffer
%TypedArray%.prototype.buffer is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let \( O \) be the this value.
2. If \( \text{Type}(O) \) is not Object, throw a TypeError exception.
3. If \( O \) does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
4. Assert: \( O \) has a [[ViewedArrayBuffer]] internal slot.
5. Let buffer be \( O.[[ViewedArrayBuffer]] \).
6. Return buffer.

### 22.2.3.2 get %TypedArray%.prototype.byteLength

%TypedArray%.prototype.byteLength is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let \( O \) be the this value.
2. If \( \text{Type}(O) \) is not Object, throw a TypeError exception.
3. If \( O \) does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
4. Assert: \( O \) has a [[ViewedArrayBuffer]] internal slot.
5. Let buffer be \( O.[[ViewedArrayBuffer]] \).
6. If IsDetachedBuffer(buffer) is true, return 0.
7. Let size be \( O.[[ByteLength]] \).
8. Return size.

### 22.2.3.3 get %TypedArray%.prototype.byteOffset

%TypedArray%.prototype.byteOffset is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let \( O \) be the this value.
2. If \( \text{Type}(O) \) is not Object, throw a TypeError exception.
3. If \( O \) does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
4. Assert: \( O \) has a [[ViewedArrayBuffer]] internal slot.
5. Let buffer be \( O.[[ViewedArrayBuffer]] \).
6. If IsDetachedBuffer(buffer) is true, return 0.
7. Let offset be \( O.[[ByteOffset]] \).
8. Return offset.

### 22.2.3.4 %TypedArray%.prototype.constructor

The initial value of %TypedArray%.prototype.constructor is the %TypedArray% intrinsic object.
22.2.3.5 `%TypedArray%.prototype.copyWithin (target, start [, end])`

The interpretation and use of the arguments of `%TypedArray%.prototype.copyWithin` are the same as for `Array.prototype.copyWithin` as defined in 22.1.3.3.

The following steps are taken:

1. Let `O` be this value.
2. Perform `? ValidateTypedArray(O)`.  
3. Let `len` be `O[[ArrayLength]]`.  
4. Let `relativeTarget` be `? ToInteger(target)`.  
5. If `relativeTarget < 0`, let `to` be `max((len + relativeTarget), 0)`; else let `to` be `min(relativeTarget, len)`.  
6. Let `relativeStart` be `? ToInteger(start)`.  
7. If `relativeStart < 0`, let `from` be `max((len + relativeStart), 0)`; else let `from` be `min(relativeStart, len)`.  
8. If `end` is `undefined`, let `relativeEnd be len`; else let `relativeEnd` be `? ToInteger(end)`.  
9. If `relativeEnd < 0`, let `final` be `max((len + relativeEnd), 0)`; else let `final` be `min(relativeEnd, len)`.  
10. Let `count` be `min(final-from, len-to)`.  
11. If `count > 0`, then
   a. NOTE: The copying must be performed in a manner that preserves the bit-level encoding of the source data.
   b. Let `buffer` be `O[[ViewedArrayBuffer]]`.
   c. If `IsDetachedBuffer(buffer)` is true, throw a `TypeError` exception.
   d. Let `typedArrayName` be the String value of `O[[TypedArrayName]]`.
   e. Let `elementSize` be the Number value of the Element Size value specified in Table 50 for `typedArrayName`.
   f. Let `byteOffset` be `O[[ByteOffset]]`.
   g. Let `toByteIndex` be `to × elementSize + byteOffset`.
   h. Let `fromByteIndex` be `from × elementSize + byteOffset`.
   i. Let `countBytes` be `count × elementSize`.
   j. If `fromByteIndex<toByteIndex` and `toByteIndex<fromByteIndex+countBytes`, then
      i. Let `direction` be `-1`.
      ii. Let `fromByteIndex` be `fromByteIndex + countBytes - 1`.
      iii. Let `toByteIndex` be `toByteIndex + countBytes - 1`.
   k. Else,
      i. Let `direction` be `1`.
   l. Repeat, while `countBytes > 0`
      i. Let `value` be `GetValueFromBuffer(buffer, fromByteIndex, "Uint8", true, "Unordered")`.
      ii. Perform `SetValueInBuffer(buffer, toByteIndex, "Uint8", value, true, "Unordered")`.
      iii. Let `fromByteIndex` be `fromByteIndex + direction`.
      iv. Let `toByteIndex` be `toByteIndex + direction`.
   v. Let `countBytes` be `countBytes - 1`.

12. Return `O`.
When called with argument \( O \), the following steps are taken:

1. If \( \text{Type}(O) \) is not \( \text{Object} \), throw a \text{TypeError} exception.
2. If \( O \) does not have a \([\text{TypedArrayName}]\) internal slot, throw a \text{TypeError} exception.
3. Assert: \( O \) has a \([\text{ViewedArrayBuffer}]\) internal slot.
4. Let \( \text{buffer} \) be \( O.\text{[ViewedArrayBuffer]} \).
5. If \( \text{isDetachedBuffer}(\text{buffer}) \) is \text{true}, throw a \text{TypeError} exception.
6. Return \( \text{buffer} \).

The following steps are taken:

1. Let \( O \) be the \text{this} value.
2. Perform \( \text{? ValidateTypedArray}(O) \).
3. Return \( \text{CreateArrayIterator}(O, \"key+value\") \).

%TypedArray%.prototype.entries ()

%TypedArray%.prototype.entries is a distinct function that implements the same algorithm as \( \text{Array.prototype.entries} \) as defined in 22.1.5 except that the \text{this} object's \([\text{ArrayLength}]\) internal slot is accessed in place of performing a \([\text{Get}]\) of \"length\". The implementation of the algorithm may be optimized with the knowledge that the \text{this} value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to \( \text{callbackfn} \) may cause the \text{this} value to become detached.

This function is not generic. \( \text{ValidateTypedArray} \) is applied to the \text{this} value prior to evaluating the algorithm. If its result is an \text{abrupt completion} that exception is thrown instead of evaluating the algorithm.

%TypedArray%.prototype.fill ( value [ , start [ , end ] ] )

%TypedArray%.prototype.fill is a distinct function that implements the same algorithm as \( \text{Array.prototype.fill} \) as defined in 22.1.6 except that the \text{this} object's \([\text{ArrayLength}]\) internal slot is accessed in place of performing a \([\text{Get}]\) of \"length\". The implementation of the algorithm may be optimized with the knowledge that the \text{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. \( \text{ValidateTypedArray} \) is applied to the \text{this} value prior to evaluating the algorithm. If its result is an \text{abrupt completion} that exception is thrown instead of evaluating the algorithm.
22.2.3.9 `%TypedArray%.prototype.filter (callbackfn [, thisArg])`

The interpretation and use of the arguments of `%TypedArray%.prototype.filter` are the same as for `Array.prototype.filter` as defined in 22.1.3.7.

When the `filter` method is called with one or two arguments, the following steps are taken:

1. Let `O` be the `this` value.
2. Perform `ValidateTypedArray(O)`.
3. Let `len` be `O.[[ArrayLength]]`.
4. If `isCallable(callbackfn)` is `false`, throw a `TypeError` exception.
5. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
6. Let `kept` be a new empty `List`.
7. Let `k` be 0.
8. Let `captured` be 0.
9. Repeat, while `k < len`
   a. Let `Pk` be `!ToString(k)`.
   b. Let `kValue` be `? Get(O, Pk)`.
   c. Let `selected` be `ToBoolean(? Call(callbackfn, T, « kValue, k, O »))`.
   d. If `selected` is `true`, then
      i. Append `kValue` to the end of `kept`.
      ii. Increase `captured` by 1.
   e. Increase `k` by 1.
10. Let `A` be `? TypedArraySpeciesCreate(O, « captured »)`.
11. Let `n` be 0.
12. For each element `e` of `kept`, do
   a. Perform `! Set(A, !ToString(n), e, true)`.
   b. Increment `n` by 1.
13. Return `A`.

This function is not generic. The `this` value must be an object with a `[[TypedArrayName]]` internal slot.

22.2.3.10 `%TypedArray%.prototype.find (predicate [, thisArg])`

`%TypedArray%.prototype.find` is a distinct function that implements the same algorithm as `Array.prototype.find` as defined in 22.1.3.8 except that the `this` object’s `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "length". The implementation of the algorithm may be optimized with the knowledge that the `this` value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to `predicate` may cause the `this` value to become detached.
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.

### 22.2.3.11 %TypedArray%.prototype.findIndex (predicate [, thisArg ])

%TypedArray%.prototype.findIndex is a distinct function that implements the same algorithm as Array.prototype.findIndex as defined in 22.1.3.9 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to predicate may cause the this value to become detached.

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.

### 22.2.3.12 %TypedArray%.prototype.forEach (callbackfn [, thisArg ])

%TypedArray%.prototype.forEach is a distinct function that implements the same algorithm as Array.prototype.forEach as defined in 22.1.3.10 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

This function is not generic. 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.

### 22.2.3.13 %TypedArray%.prototype.includes (searchElement [, fromIndex ])

%TypedArray%.prototype.includes is a distinct function that implements the same algorithm as Array.prototype.includes as defined in 22.1.3.11 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. 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.
22.2.3.14 %TypedArray%.prototype.indexOf (searchElement [, fromIndex])

%TypedArray%.prototype.indexOf is a distinct function that implements the same algorithm as Array.prototype.indexOf as defined in 22.1.3.12 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. 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.

22.2.3.15 %TypedArray%.prototype.join (separator)

%TypedArray%.prototype.join is a distinct function that implements the same algorithm as Array.prototype.join as defined in 22.1.3.13 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. 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.

22.2.3.16 %TypedArray%.prototype.keys ()

The following steps are taken:

1. Let O be the this value.
2. Perform ? ValidateTypedArray(O).
3. Return CreateArrayIterator(O, "key").

22.2.3.17 %TypedArray%.prototype.lastIndexOf (searchElement [, fromIndex])

%TypedArray%.prototype.lastIndexOf is a distinct function that implements the same algorithm as Array.prototype.lastIndexOf as defined in 22.1.3.15 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. 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.
22.2.3.18 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. If Type(\( O \)) is not Object, throw a TypeError exception.
3. If \( O \) does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
4. Assert: \( O \) has [[ViewedArrayBuffer]] and [[ArrayLength]] internal slots.
5. Let buffer be \( O.\) [[ViewedArrayBuffer]].
6. If IsDetachedBuffer(buffer) is true, return 0.
7. Let length be \( O.\) [[ArrayLength]].
8. Return length.

This function is not generic. The this value must be an object with a [[TypedArrayName]] internal slot.

22.2.3.19 %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 22.1.16.

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. If thisArg was supplied, let \( T \) be thisArg; else let \( T \) be undefined.
6. Let A be ? TypedArraySpeciesCreate(\( O \), « len »).
7. Let \( k \) be 0.
8. Repeat, while \( k < len \)
   a. Let \( Pk \) be ! ToString(\( k \)).
   b. Let \( kValue \) be ? Get(\( O \), \( Pk \)).
   c. Let \( mappedValue \) be ? Call(callbackfn, \( T \), « kValue, \( k \), \( O \) »).
   d. Perform ? Set(\( A \), \( Pk \), \( mappedValue \), true).
   e. Increase \( k \) by 1.
9. Return A.

This function is not generic. The this value must be an object with a [[TypedArrayName]] internal slot.

22.2.3.20 %TypedArray%.prototype.reduce (callbackfn [, initialValue ])


%TypedArray%.prototype.reduce is a distinct function that implements the same algorithm as Array.prototype.reduce as defined in 22.1.3.19 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

This function is not generic. 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.

22.2.3.21 %TypedArray%.prototype.reduceRight ( callbackfn [, initialValue ] )

%TypedArray%.prototype.reduceRight is a distinct function that implements the same algorithm as Array.prototype.reduceRight as defined in 22.1.3.20 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

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.

22.2.3.22 %TypedArray%.prototype.reverse ( )

%TypedArray%.prototype.reverse is a distinct function that implements the same algorithm as Array.prototype.reverse as defined in 22.1.3.21 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. 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.

22.2.3.23 %TypedArray%.prototype.set ( overloaded [, offset ] )

%TypedArray%.prototype.set is a single function whose behaviour is overloaded 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.
22.2.3.23.1 %TypedArray%.prototype.set( array [, offset ] )

Sets multiple values in this TypedArray, reading the values from the object array. The optional offset value indicates the first element index in this TypedArray where values are written. If omitted, it is assumed to be 0.

1. Assert: array is any ECMAScript language value other than an Object with a [[TypedArrayName]] internal slot. If it is such an Object, the definition in 22.2.3.23.2 applies.
2. Let target be the this value.
3. If Type(target) is not Object, throw a TypeError exception.
4. If target does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
5. Assert: target has a [[ViewedArrayBuffer]] internal slot.
7. If targetOffset < 0, throw a RangeError exception.
8. Let targetBuffer be target. [[ViewedArrayBuffer]].
9. If IsDetachedBuffer(targetBuffer) is true, throw a TypeError exception.
10. Let targetLength be target. [[ArrayLength]].
11. Let targetName be the String value of target. [[TypedArrayName]].
12. Let targetElementSize be the Number value of the Element Size value specified in Table 50 for targetName.
13. Let targetType be the String value of the Element Type value in Table 50 for targetName.
14. Let targetByteOffset be target. [[ByteOffset]].
15. Let src be ? ToObject(array).
16. Let srcLength be ? ToLength(? Get(src, "length")).
17. If srcLength + targetOffset > targetLength, throw a RangeError exception.
18. Let targetByteIndex be targetOffset × targetElementSize + targetByteOffset.
19. Let k be 0.
20. Let limit be targetByteIndex + targetElementSize × srcLength.
21. Repeat, while targetByteIndex < limit
   a. Let Pk be ! ToString(k).
   b. Let kNumber be ? ToNumber(? Get(src, Pk)).
   c. If IsDetachedBuffer(targetBuffer) is true, throw a TypeError exception.
   d. Perform SetValueInBuffer(targetBuffer, targetByteIndex, targetType, kNumber, true, "Unordered").
   e. Set k to k + 1.
   f. Set targetByteIndex to targetByteIndex + targetElementSize.
22. Return undefined.

22.2.3.23.2 %TypedArray%.prototype.set( typedArray [, offset ] )

Sets multiple values in this TypedArray, reading the values from the typedArray argument object. The optional offset value indicates the first element index in this TypedArray where values are written. If
omitted, it is assumed to be 0.

1. Assert: `typedArray` has a `[[TypedArrayName]]` internal slot. If it does not, the definition in 22.2.3.23.1 applies.
2. Let `target` be the this value.
3. If `Type(target)` is not Object, throw a `TypeError` exception.
4. If `target` does not have a `[[TypedArrayName]]` internal slot, throw a `TypeError` exception.
5. Assert: `target` has a `[[ViewedArrayBuffer]]` internal slot.
6. Let `targetOffset` be ? `ToInteger(offset)`.
7. If `targetOffset < 0`, throw a `RangeError` exception.
8. Let `targetBuffer` be `target`.`[[ViewedArrayBuffer]]`.
9. If `IsDetachedBuffer(targetBuffer)` is `true`, throw a `TypeError` exception.
10. Let `targetLength` be `target`.`[[ArrayLength]]`.
11. Let `srcBuffer` be `typedArray`.`[[ViewedArrayBuffer]]`.
12. If `IsDetachedBuffer(srcBuffer)` is `true`, throw a `TypeError` exception.
13. Let `targetName` be the String value of `target`.`[[TypedArrayName]]`.
14. Let `targetType` be the String value of the Element Type value in Table 50 for `targetName`.
15. Let `targetElementSize` be the Number value of the Element Size value specified in Table 50 for `targetName`.
16. Let `targetByteOffset` be `target`.`[[ByteOffset]]`.
17. Let `srcName` be the String value of `typedArray`.`[[TypedArrayName]]`.
18. Let `srcType` be the String value of the Element Type value in Table 50 for `srcName`.
19. Let `srcElementSize` be the Number value of the Element Size value specified in Table 50 for `srcName`.
20. Let `srcLength` be `typedArray`.`[[ArrayLength]]`.
21. Let `srcByteOffset` be `typedArray`.`[[ByteOffset]]`.
22. If `srcLength + targetOffset > targetLength`, throw a `RangeError` exception.
23. 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`.
24. Else, let `same` be `SameValue(srcBuffer, targetBuffer)`.
25. If `same` is `true`, then
   a. Let `srcByteLength` be `typedArray`.`[[ByteLength]]`.
   b. Let `srcBuffer` be ? `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.
26. Else, let `srcByteIndex` be `srcByteOffset`.
27. Let `targetByteIndex` be `targetOffset × targetElementSize + targetByteOffset`.
28. Let `limit` be `targetByteIndex + targetElementSize × srcLength`.
29. If `SameValue(srcType, targetType)` is `true`, 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 targetType < 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.

30. 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.

31. Return undefined.

22.2.3.24 %TypedArray%.prototype.slice (start, end)

The interpretation and use of the arguments of %TypedArray%.prototype.slice are the same as for Array.prototype.slice as defined in 22.1.3.23. The following steps are taken:

1. Let O be the this value.
2. Perform ? ValidateTypedArray(O).
3. Let len be O.[[ArrayLength]].
4. Let relativeStart be ? ToInteger(start).
5. If relativeStart < 0, let k be max((len + relativeStart, 0)); else let k be min(relativeStart, len).
6. If end is undefined, let relativeEnd be len; else let relativeEnd be ? ToInteger(end).
7. If relativeEnd < 0, let final be max((len + relativeEnd, 0)); else let final be min(relativeEnd, len).
8. Let count be max(final - k, 0).
9. Let A be ? TypedArraySpeciesCreate(O, « count »).
10. Let srcName be the String value of O.[[TypedArrayName]].
11. Let srcType be the String value of the Element Type value in Table 50 for srcName.
12. Let targetName be the String value of A.[[TypedArrayName]].
13. Let targetType be the String value of the Element Type value in Table 50 for targetName.
14. If SameValue(srcType, targetType) is false, then
   a. Let n be 0.
   b. Repeat, while k < final
      i. Let Pk be ! ToString(k).
      ii. Let kValue be ? Get(O, Pk).
      iii. Perform ! Set(A, ! ToString(n), kValue).
iv. Increase \( k \) by 1.

v. Increase \( n \) by 1.

15. Else if \( \text{count} > 0 \), then

a. Let \( \text{srcBuffer} \) be \( \text{O.}[[\text{ViewedArrayBuffer}]] \).

b. If \( \text{IsDetachedBuffer(srcBuffer)} \) is \text{true}, throw a \text{TypeError} exception.

c. Let \( \text{targetBuffer} \) be \( \text{A.}[[\text{ViewedArrayBuffer}]] \).

d. Let \( \text{elementSize} \) be the Number value of the Element Size value specified in \text{Table 50} for \( \text{srcType} \).

e. \text{NOTE: If } \text{srcType} \text{ and } \text{targetType} \text{ are the same, the transfer must be performed in a manner that preserves the bit-level encoding of the source data.}

f. Let \( \text{srcByteOffset} \) be \( \text{O.}[[\text{ByteOffset}]] \).

g. Let \( \text{targetByteIndex} \) be \( \text{A.}[[\text{ByteOffset}]] \).

h. Let \( \text{srcIndex} \) be \( (k \times \text{elementSize}) + \text{srcByteOffset} \).

i. Let \( \text{limit} \) be \( \text{targetByteIndex} + \text{count} \times \text{elementSize} \).

j. Repeat, while \( \text{targetByteIndex} < \text{limit} \)

   i. Let \( \text{value} \) be \( \text{GetValueFromBuffer(srcBuffer, srcByteIndex, "Uint8", true, "Unordered").} \)

   ii. Perform \( \text{SetValueInBuffer(targetBuffer, targetByteIndex, "Uint8", value, true, "Unordered")}. \)

   iii. Increase \( \text{srcByteIndex} \) by 1.

   iv. Increase \( \text{targetByteIndex} \) by 1.

16. Return \( \text{A} \).

This function is not generic. The \text{this} value must be an object with a \([\text{TypedArrayName}]\) internal slot.

\text{22.2.3.25 \%TypedArray\%\text{.prototype.some ( callbckfn [ , thisArg ] )}}

\%TypedArray\%.\text{prototype.some} is a distinct function that implements the same algorithm as \text{Array\%.prototype.some} as defined in \text{22.1.3.24} except that the \text{this} object’s \([\text{ArrayLength}]\) internal slot is accessed in place of performing a \([\text{Get}]\) of \text{"length"}. The implementation of the algorithm may be optimized with the knowledge that the \text{this} value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to \text{callbackfn} may cause the \text{this} value to become detached.

This function is not generic. \text{ValidateTypedArray} is applied to the \text{this} value prior to evaluating the algorithm. If its result is an \text{abrupt completion} that exception is thrown instead of evaluating the algorithm.

\text{22.2.3.26 \%TypedArray\%\text{.prototype.sort ( comparefn )}}

\%TypedArray\%.\text{prototype.sort} is a distinct function that, except as described below, implements the same requirements as those of \text{Array\%.prototype.sort} as defined in \text{22.1.3.25}. The implementation of the \%TypedArray\%.\text{prototype.sort} specification may be optimized with the knowledge that the \text{this}
value is an object that has a fixed length and whose integer indexed properties are not sparse. The only internal methods of the this object that the algorithm may call are [[Get]] and [[Set]].

This function is not generic. The this value must be an object with a [[TypedArrayName]] internal slot.

Upon entry, the following steps are performed to initialize evaluation of the sort function. These steps are used instead of the entry steps in 22.1.3.25:

1. Let \( \text{obj} \) be the this value.
2. Let \( \text{buffer} \) be \( \text{ValidateTypedArray}(\text{obj}) \).
3. Let \( \text{len} \) be \( \text{obj}.[[\text{ArrayLength}]] \).

The implementation-defined sort order condition for exotic objects is not applied by %TypedArray%.prototype.sort.

The following version of SortCompare is used by %TypedArray%.prototype.sort. It performs a numeric comparison rather than the string comparison used in 22.1.3.25. SortCompare has access to the comparefn and buffer values of the current invocation of the sort method.

When the TypedArray SortCompare abstract operation is called with two arguments \( x \) and \( y \), the following steps are taken:

1. Assert: Both Type(\( x \)) and Type(\( y \)) is Number.
2. If the argument comparefn is not undefined, then
   a. Let \( v \) be ? Call(comparefn, undefined, \( \llbracket x, y \rrbracket \)).
   b. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
   c. If \( v \) is NaN, return +0.
   d. Return \( v \).
3. If \( x \) and \( y \) are both NaN, return +0.
4. If \( x \) is NaN, return 1.
5. If \( y \) is NaN, return -1.
6. If \( x < y \), return -1.
7. If \( x > y \), return 1.
8. If \( x \) is -0 and \( y \) is +0, return -1.
9. If \( x \) is +0 and \( y \) is -0, return 1.
10. Return +0.

NOTE Because NaN always compares greater than any other value, NaN property values always sort to the end of the result when comparefn is not provided.

22.2.3.27 %TypedArray%.prototype.subarray( begin, end )
Returns a new *TypedArray* object whose element type is the same as this *TypedArray* and whose ArrayBuffer is the same as the ArrayBuffer of this *TypedArray*, referencing the elements at `begin`, inclusive, up to `end`, exclusive. If either `begin` or `end` is negative, it refers to an index from the end of the array, as opposed to from the beginning.

1. Let `O` be the *this* value.
2. If `Type(O)` is not Object, throw a *TypeError* exception.
3. If `O` does not have a `[[TypedArrayName]]` internal slot, throw a *TypeError* exception.
4. Assert: `O` has a `[[ViewedArrayBuffer]]` internal slot.
5. Let `buffer` be `O. [[ViewedArrayBuffer]]`.
7. Let `relativeBegin` be `? ToInteger(begin)`.
8. If `relativeBegin < 0`, let `beginIndex` be `max((srcLength + relativeBegin), 0)`; else let `beginIndex` be `min(relativeBegin, srcLength)`.
9. If `end` is `undefined`, let `relativeEnd` be `srcLength`; else, let `relativeEnd` be `? ToInteger(end)`.
10. If `relativeEnd < 0`, let `endIndex` be `max((srcLength + relativeEnd), 0)`; else let `endIndex` be `min(relativeEnd, srcLength)`.
11. Let `newLength` be `max(endIndex - beginIndex, 0)`.
12. Let `constructorName` be the String value of `O. [[TypedArrayName]]`.
13. Let `elementSize` be the Number value of the Element Size value specified in *Table 50* for `constructorName`.
14. Let `srcByteOffset` be `O. [[ByteOffset]]`.
15. Let `beginByteOffset` be `srcByteOffset + beginIndex × elementSize`.
16. Let `argumentsList` be « `buffer`, `beginByteOffset`, `newLength` ».
17. Return `? TypedArraySpeciesCreate(O, argumentsList)`.

This function is not generic. The *this* value must be an object with a `[[TypedArrayName]]` internal slot.

### 22.2.3.28 %TypedArray% . prototype . toLocaleString ( [ `reserved1` [ , `reserved2` ] ] )

%TypedArray% . *prototype* . `toLocaleString` is a distinct function that implements the same algorithm as `Array.prototype.toLocaleString` as defined in 22.1.3.27 except that the *this* object’s `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "%length". The implementation of the algorithm may be optimized with the knowledge that the *this* value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. `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
22.2.3.29 %TypedArray%.prototype.toString()

The initial value of the %TypedArray%.prototype.toString data property is the same built-in function object as the Array.prototype.toString method defined in 22.1.3.28.

22.2.3.30 %TypedArray%.prototype.values()

The following steps are taken:

1. Let O be the this value.
2. Perform ? ValidateTypedArray(O).
3. Return CreateArrayIterator(O, "value").

22.2.3.31 %TypedArray%.prototype @@iterator() ()

The initial value of the @@iterator property is the same function object as the initial value of the %TypedArray%.prototype.values property.

22.2.3.32 get %TypedArray%.prototype @@toStringTag

%TypedArray%.prototype[@@toStringTag] is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, return undefined.
3. If O does not have a [[TypedArrayName]] internal slot, return undefined.
4. Let name be O.[[TypedArrayName]].
5. Assert: name is a String value.
6. Return name.

This property has the attributes { [[Enumerable]]: false, [[Configurable]]: true }.

The initial value of the name property of this function is "get [Symbol.toStringTag]".

22.2.4 The TypedArray Constructors

Each of the TypedArray constructor objects 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 50.
The *TypedArray* intrinsic constructor functions are single functions whose behaviour is overloaded 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.

The *TypedArray* constructors are not intended to be called as a function and will throw an exception when called in that manner.

The *TypedArray* constructors are designed to be subclassable. They 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.

The `length` property of the *TypedArray* constructor function is 3.

### 22.2.4.1 *TypedArray* ()

This description applies only if the *TypedArray* function is called with no arguments.

1. If `NewTarget` is `undefined`, throw a `TypeError` exception.
2. Let `constructorName` be the String value of the Constructor Name value specified in Table 50 for this `TypedArray` constructor.
3. Return `? AllocateTypedArray(constructorName, NewTarget, "%TypedArrayPrototype", 0)`.

### 22.2.4.2 *TypedArray* (length)

This description applies only if the *TypedArray* function is called with at least one argument and the Type of the first argument is not `Object`.

*TypedArray* called with argument `length` performs the following steps:

1. Assert: `Type(length)` is not `Object`.
2. If `NewTarget` is `undefined`, throw a `TypeError` exception.
3. Let `elementLength` be `? ToIndex(length)`.
4. Let `constructorName` be the String value of the Constructor Name value specified in Table 50 for this `TypedArray` constructor.

### 22.2.4.2.1 Runtime Semantics: AllocateTypedArray (constructorName, newTarget, defaultProto [ , length ])

The abstract operation `AllocateTypedArray` with arguments `constructorName, newTarget, defaultProto` and optional argument `length` is used to validate and create an instance of a *TypedArray* constructor.
constructorName is required to be the name of a TypedArray constructor in Table 50. 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 all of the TypedArray overloads. AllocateTypedArray performs the following steps:

1. Let proto be ? GetPrototypeFromConstructor(newTarget, defaultProto).
2. Let obj be IntegerIndexedObjectCreate(proto, « [[ViewedArrayBuffer]], [[TypedArrayName]], [[ByteLength]], [[ByteOffset]], [[ArrayLength]] »).
3. Assert: obj.[[ViewedArrayBuffer]] is undefined.
4. Set obj.[[TypedArrayName]] to constructorName.
5. If length was not passed, then
   a. Set obj.[[ByteLength]] to 0.
   b. Set obj.[[ByteOffset]] to 0.
   c. Set obj.[[ArrayLength]] to 0.
6. Else,
   a. Perform ? AllocateTypedArrayBuffer(obj, length).
7. Return obj.

22.2.4.2.2 Runtime Semantics: AllocateTypedArrayBuffer (O, length)

The abstract operation AllocateTypedArrayBuffer with arguments O and length allocates and associates an ArrayBuffer with the TypedArray instance O. It performs the following steps:

1. Assert: O is an Object that has a [[ViewedArrayBuffer]] internal slot.
2. Assert: O.[[ViewedArrayBuffer]] is undefined.
3. Assert: length \( \geq 0 \).
4. Let constructorName be the String value of O.[[TypedArrayName]].
5. Let elementSize be the Element Size value in Table 50 for constructorName.
6. Let byteLength be elementSize \( \times \) length.
7. Let data be ? AllocateArrayBuffer(%ArrayBuffer%, byteLength).
8. Set O.[[ViewedArrayBuffer]] to data.
10. Set O.[[ByteOffset]] to 0.
11. Set O.[[ArrayLength]] to length.
12. Return O.

22.2.4.3 TypedArray (typedArray)

This description applies only if the TypedArray function is called with at least one argument and the Type of the first argument is Object and that object has a [[TypedArrayName]] internal slot.

TypedArray called with argument typedArray performs the following steps:
1. Assert: `Type(typedArray)` is Object and `typedArray` has a `[[TypedArrayName]]` internal slot.
2. If `NewTarget` is `undefined`, throw a `TypeError` exception.
3. Let `constructorName` be the String value of the Constructor Name value specified in Table 50 for this `TypedArray` constructor.
4. Let `O` be `AllocateTypedArray(constructorName, NewTarget, "%TypedArrayPrototype%")`.
5. Let `srcArray` be `typedArray`.[[ViewedArrayBuffer]].
6. Let `srcData` be `srcArray`.[[ViewedArrayBuffer]].
7. If `IsDetachedBuffer(srcData)` is `true`, throw a `TypeError` exception.
8. Let `elementType` be the String value of the Element Type value in Table 50 for `constructorName`.
9. Let `elementLength` be `srcArray`.[[ArrayLength]].
10. Let `srcName` be the String value of `srcArray`.[[TypedArrayName]].
11. Let `srcType` be the String value of the Element Type value in Table 50 for `srcName`.
12. Let `srcElementSize` be the Element Size value in Table 50 for `srcName`.
13. Let `srcByteOffset` be `srcArray`.[[ByteOffset]].
14. Let `elementSize` be the Element Size value in Table 50 for `constructorName`.
15. Let `byteLength` be `elementSize × elementLength`.
16. If `SameValue(elementType, srcType)` is `true`, then
   a. If `IsSharedArrayBuffer(srcData)` is `false`, then
      i. Let `data` be `CloneArrayBuffer(srcData, srcByteOffset, byteLength)`.
   b. Else,
      i. Let `data` be `CloneArrayBuffer(srcData, srcByteOffset, byteLength, %ArrayBuffer%)
17. Else,
   a. If `IsSharedArrayBuffer(srcData)` is `false`, then
      i. Let `bufferConstructor` be `SpeciesConstructor(srcData, %ArrayBuffer%)
   b. Else,
      i. Let `bufferConstructor` be `%ArrayBuffer%.
   c. Let `data` be `AllocateArrayBuffer(bufferConstructor, byteLength)
   d. If `IsDetachedBuffer(srcData)` is `true`, throw a `TypeError` exception.
   e. Let `srcByteIndex` be `srcByteOffset`.
   f. Let `targetByteIndex` be `0`.
   g. Let `count` be `elementLength`.
   h. Repeat, while `count > 0`
      i. Let `value` be `GetValueFromBuffer(srcData, srcByteIndex, srcType, true, "Unordered")
      ii. Perform `SetValueInBuffer(data, targetByteIndex, elementType, value, true, "Unordered")`.
      iii. Set `srcByteIndex` to `srcByteIndex + srcElementSize`.
      iv. Set `targetByteIndex` to `targetByteIndex + elementSize`.
      v. Decrement `count` by 1.
18. Set `O`.[[ViewedArrayBuffer]] to `data`.
19. Set `O`.[[ByteLength]] to `byteLength`. 
20. Set $O[[\text{ByteOffset}]]$ to 0.
21. Set $O[[\text{ArrayLength}]]$ to $\text{elementLength}$.
22. Return $O$.

### 22.2.4.4 $\textbf{TypedArray ( object )}$

This description applies only if the $\text{TypedArray}$ function is called with at least one argument and the Type of the first argument is $\text{Object}$ and that object does not have either a $[[\text{TypedArrayName}]]$ or an $[[\text{ArrayBufferData}]]$ internal slot.

$\text{TypedArray}$ called with argument $\text{object}$ performs the following steps:

1. Assert: $\text{Type( object )}$ is $\text{Object}$ and $\text{object}$ does not have either a $[[\text{TypedArrayName}]]$ or an $[[\text{ArrayBufferData}]]$ internal slot.
2. If $\text{NewTarget}$ is undefined, throw a $\text{TypeError}$ exception.
3. Let $\text{constructorName}$ be the String value of the Constructor Name value specified in Table 50 for this $\text{TypedArray}$ constructor.
4. Let $O$ be $\text{AllocateTypedArray( constructorName, NewTarget, "%TypedArrayPrototype%")}$. 
5. Let $\text{usingIterator}$ be $\text{GetMethod( object, @@iterator )}$. 
6. If $\text{usingIterator}$ is not undefined, then
   a. Let $\text{values}$ be $\text{IterableToList( object, usingIterator )}$.
   b. Let $\text{len}$ be the number of elements in $\text{values}$.
   c. Perform $\text{AllocateTypedArrayBuffer( O, len )}$.
   d. Let $k$ be 0.
   e. Repeat, while $k < \text{len}$
      i. Let $Pk$ be $\text{ToString( k )}$.
      ii. Let $kValue$ be the first element of $\text{values}$ and remove that element from $\text{values}$.
      iii. Perform $\text{Set( O, Pk, kValue, true )}$.
      iv. Increase $k$ by 1.
   f. Assert: $\text{values}$ is now an empty List.
   g. Return $O$.
7. NOTE: $\text{object}$ is not an Iterable so assume it is already an array-like object.
8. Let $\text{arrayLike}$ be $\text{object}$.
9. Let $\text{len}$ be $\text{MaxLength( ? Get(arrayLike, "length") )}$.
10. Perform $\text{AllocateTypedArrayBuffer( O, len )}$.
11. Let $k$ be 0.
12. Repeat, while $k < \text{len}$
   a. Let $Pk$ be $\text{ToString( k )}$.
   b. Let $kValue$ be $\text{Get( arrayLike, Pk )}$.
   c. Perform $\text{Set( O, Pk, kValue, true )}$.
   d. Increase $k$ by 1.
13. Return $O$.

### 22.2.4.5 `TypedArray ( buffer [, byteOffset [, length ]] )`

This description applies only if the `TypedArray` function is called with at least one argument and the Type of the first argument is Object and that object has an `[[ArrayBufferData]]` internal slot.

`TypedArray` called with at least one argument `buffer` performs the following steps:

1. Assert: `Type(buffer)` is Object and `buffer` has an `[[ArrayBufferData]]` internal slot.
2. If NewTarget is `undefined`, throw a `TypeError` exception.
3. Let `constructorName` be the String value of the Constructor Name value specified in Table 50 for this `TypedArray` constructor.
4. Let $O$ be `AllocateTypedArray(constructorName, NewTarget, "%TypedArrayPrototype").`
5. Let `elementSize` be the Number value of the Element Size value in Table 50 for `constructorName`.
6. Let `offset` be `ToIndex(byteOffset)`.
7. If `offset` modulo `elementSize` ≠ 0, throw a `RangeError` exception.
8. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
9. Let `bufferByteLength` be `buffer.[[ArrayBufferByteLength]]`.
10. If `length` is either not present or `undefined`, then
    a. If `bufferByteLength` modulo `elementSize` ≠ 0, throw a `RangeError` exception.
    b. Let `newByteLength` be `bufferByteLength - offset`.
    c. If `newByteLength < 0`, throw a `RangeError` exception.
11. Else,
    a. Let `newLength` be `ToIndex(length)`.
    b. Let `newByteLength` be `newLength × elementSize`.
    c. If `offset + newByteLength > bufferByteLength`, throw a `RangeError` exception.
12. Set $O.[[ViewedArrayBuffer]]$ to `buffer`.
13. Set $O.[[ByteLength]]$ to `newByteLength`.
14. Set $O.[[ByteOffset]]$ to `offset`.
15. Set $O.[[ArrayLength]]$ to `newByteLength / elementSize`.

### 22.2.4.6 `TypedArrayCreate ( constructor, argumentList )`

The abstract operation `TypedArrayCreate` with arguments `constructor` and `argumentList` is used to specify the creation of a new TypedArray object using a constructor function. It performs the following steps:

1. Let `newTypedArray` be `? Construct(constructor, argumentList)`.
2. Perform `? ValidateTypedArray(newTypedArray)`.
3. If `argumentList` is a List of a single Number, then
   a. If `newTypedArray.[[ArrayLength]] < argumentList[0]`, throw a `TypeError` exception.
4. Return `newTypedArray`.

### 22.2.4.7 `TypedArraySpeciesCreate (exemplar, argumentList)`

The abstract operation `TypedArraySpeciesCreate` with arguments `exemplar` and `argumentList` 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:

1. Assert: `exemplar` is an Object that has a `[[TypedArrayName]]` internal slot.
2. Let `defaultConstructor` be the intrinsic object listed in column one of Table 50 for `exemplar`. `[[TypedArrayName]]`.
3. Let `constructor` be `? SpeciesConstructor(exemplar, defaultConstructor)`.
4. Return `? TypedArrayCreate(constructor, argumentList)`.

### 22.2.5 Properties of the `TypedArray` Constructors

The value of the `[[Prototype]]` internal slot of each `TypedArray` constructor is the `%TypedArray%` intrinsic object.

Each `TypedArray` constructor has a `name` property whose value is the String value of the constructor name specified for it in Table 50.

Each `TypedArray` constructor has the following properties:

#### 22.2.5.1 `TypedArray.BYTES_PER_ELEMENT`

The value of `TypedArray.BYTES_PER_ELEMENT` is the Number value of the Element Size value specified in Table 50 for `TypedArray`.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

#### 22.2.5.2 `TypedArray.prototype`

The initial value of `TypedArray.prototype` is the corresponding `TypedArray` prototype intrinsic object (22.2.6).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 22.2.6 Properties of `TypedArray` Prototype Objects

The value of the `[[Prototype]]` internal slot of a `TypedArray` prototype object is the intrinsic object `%TypedArrayPrototype%`. A `TypedArray` prototype object is an ordinary object. It does not have a
[ViewedArrayBuffer]] or any other of the internal slots that are specific to *TypedArray* instance objects.

### 22.2.6.1 *TypedArray*.prototype.BYTES_PER_ELEMENT

The value of *TypedArray*.prototype.BYTES_PER_ELEMENT is the Number value of the Element Size value specified in Table 50 for *TypedArray*.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 22.2.6.2 *TypedArray*.prototype.constructor

The initial value of a *TypedArray*.prototype.constructor is the corresponding `%TypedArray% intrinsic object.

### 22.2.7 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]].

## 23 Keyed Collection

### 23.1 Map Objects

Map objects are collections of key/value pairs where both the keys and values may be arbitrary ECMAScript language values. A distinct key value may only occur in one key/value pair within the Map’s collection. Distinct key values are discriminated using the SameValueZero comparison algorithm.

Map object must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Map objects specification is only intended to describe the required observable semantics of Map objects. It is not intended to be a viable implementation model.

### 23.1.1 The Map Constructor

The Map constructor is the `%Map% intrinsic object and the initial value of the Map property of the global object. When called as a constructor it creates and initializes a new Map object. Map is not intended to be called as a function and will throw an exception when called in that manner.
The Map constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified Map behaviour must include a `super` call to the Map constructor to create and initialize the subclass instance with the internal state necessary to support the Map.prototype built-in methods.

### 23.1.1.1 Map ([`iterable`])

When the Map function is called with optional argument, the following steps are taken:

1. If `NewTarget` is `undefined`, throw a `TypeError` exception.
2. Let `map` be `? OrdinaryCreateFromConstructor(NewTarget, "%MapPrototype%", « [[MapData]] »).
3. Set `map.[[MapData]]` to a new empty `List`.
4. If `iterable` is not present, let `iterable` be `undefined`.
5. If `iterable` is either `undefined` or `null`, let `iter` be `undefined`.
6. Else,
   a. Let `adder` be `Get(map, "set")`.
   b. If `IsCallable(adder)` is `false`, throw a `TypeError` exception.
   c. Let `iter` be `? GetIterator(iterable)`.
7. If `iter` is `undefined`, return `map`.
8. Repeat,
   a. Let `next` be `? IteratorStep(iter)`.
   b. If `next` is `false`, return `map`.
   c. Let `nextItem` be `? IteratorValue(next)`.
   d. If `Type(nextItem)` is not `Object`, then
      i. Let `error` be `Completion{[[Type]]: throw, [[Value]]: a newly created TypeError object, [[Target]]: empty}`.
      ii. Return `? IteratorClose(iter, error)`.
   e. Let `k` be `Get(nextItem, "0")`.
   f. If `k` is an abrupt completion, return `? IteratorClose(iter, k)`.
   g. Let `v` be `Get(nextItem, "1")`.
   h. If `v` is an abrupt completion, return `? IteratorClose(iter, v)`.
      i. Let `status` be `Call(adder, map, « k.[[Value]], v.[[Value]] »)`.
      j. If `status` is an abrupt completion, return `? IteratorClose(iter, status)`.

**NOTE** If the parameter `iterable` is present, it is expected to be an object that implements an `@@iterator` method that returns an iterator object that produces a two element array-like object whose first element is a value that will be used as a Map key and whose second element is the value to associate with that key.

### 23.1.2 Properties of the Map Constructor
The value of the [[Prototype]] internal slot of the Map constructor is the intrinsic object %FunctionPrototype%.

The Map constructor has the following properties:

### 23.1.2.1 Map.prototype

The initial value of `Map.prototype` is the intrinsic object %MapPrototype%.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 23.1.2.2 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.

### 23.1.3 Properties of the Map Prototype Object

The Map prototype object is the intrinsic object %MapPrototype%. The value of the [[Prototype]] internal slot of the Map prototype object is the intrinsic object %ObjectPrototype%. The Map prototype object is an ordinary object. It does not have a [[MapData]] internal slot.

### 23.1.3.1 Map.prototype.clear()

The following steps are taken:

1. Let `M` be the `this` value.
2. If `Type(M)` is not `Object`, throw a `TypeError` exception.
3. If `M` does not have a [[MapData]] internal slot, throw a `TypeError` exception.
4. Let `entries` be the `List` that is `M.[[MapData]]`.
5. For each `Record` `[[[Key]], [[Value]]] p` that is an element of `entries`, do
   a. Set `p.[[Key]]` to empty.
   b. Set `p.[[Value]]` to empty.
6. 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`.

### 23.1.3.2 Map.prototype.constructor

The initial value of `Map.prototype.constructor` is the intrinsic object `%Map%`.

### 23.1.3.3 Map.prototype.delete ( `key` )

The following steps are taken:

1. Let `M` be the `this` value.
2. If `Type(M)` is not Object, throw a `TypeError` exception.
3. If `M` does not have a `[[MapData]]` internal slot, throw a `TypeError` exception.
4. Let `entries` be the `List` that is `M`. `[[MapData]]`.
5. For each `Record `[[Key]], [[Value]]` `p` that is an element 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`.
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.

### 23.1.3.4 Map.prototype.entries ( )

The following steps are taken:

1. Let `M` be the `this` value.
2. Return `? CreateMapIterator(M, "key+value")`.

### 23.1.3.5 Map.prototype.forEach ( `callbackfn` [], `thisArg` )

When the `forEach` method is called with one or two arguments, the following steps are taken:

1. Let `M` be the `this` value.
2. If `Type(M)` is not Object, throw a `TypeError` exception.
3. If `M` does not have a `[[MapData]]` internal slot, throw a `TypeError` exception.
4. If `Callable(callbackfn)` is `false`, throw a `TypeError` exception.
5. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`. 
6. Let entries be the List that is $M$.[[MapData]].
7. For each Record $\{[[\text{Key}], \text{[[Value]]}]\} \ e$ that is an element of entries, in original key insertion order, do
   a. If $e.\text{[[Key]]}$ is not empty, then
      i. Perform ? Call($callbackfn$, $T$, « $e.\text{[[Value]]}$, $e.\text{[[Key]]}$, $M$ »).
8. 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.

23.1.3.6 Map.prototype.get (key)

The following steps are taken:

1. Let $M$ be the this value.
2. If Type($M$) is not Object, throw a TypeError exception.
3. If $M$ does not have a [[MapData]] internal slot, throw a TypeError exception.
4. Let entries be the List that is $M$.[[MapData]].
5. For each Record $\{[[\text{Key}], \text{[[Value]]}]\} \ p$ that is an element of entries, do
   a. If $p.\text{[[Key]]}$ is not empty and SameValueZero($p.\text{[[Key]]}$, $key$) is true, return $p.\text{[[Value]]}$.
6. Return undefined.

23.1.3.7 Map.prototype.has (key)

The following steps are taken:

1. Let $M$ be the this value.
2. If Type($M$) is not Object, throw a TypeError exception.
3. If $M$ does not have a [[MapData]] internal slot, throw a TypeError exception.
4. Let \textit{entries} be the \texttt{List} that is \textit{M}.[[\text{MapData}]].

5. For each \texttt{Record} \{[[\text{Key}]], [[\text{Value}]]\} \textit{p} that is an element of \textit{entries}, do
   a. If \textit{p}.[[\text{Key}]] is not empty and \texttt{SameValueZero}(\textit{p}.[[\text{Key}]], \textit{key}) is \texttt{true}, return \texttt{true}.
   b. If \textit{p}.[[\text{Key}]] is empty, return \texttt{false}.

6. Return \texttt{false}.

\subsection*{23.1.3.8 Map.prototype.keys ( )}

The following steps are taken:

1. Let \textit{M} be the \texttt{this} value.
2. Return ? \texttt{CreateMapIterator}(\textit{M}, "key").

\subsection*{23.1.3.9 Map.prototype.set ( \texttt{key, value} )}

The following steps are taken:

1. Let \textit{M} be the \texttt{this} value.
2. If \texttt{Type}(\textit{M}) is not \texttt{Object}, throw a \texttt{TypeError} exception.
3. If \textit{M} does not have a [[\text{MapData}]] internal slot, throw a \texttt{TypeError} exception.
4. Let \textit{entries} be the \texttt{List} that is \textit{M}.[[\text{MapData}]].
5. For each \texttt{Record} \{[[\text{Key}]], [[\text{Value}]]\} \textit{p} that is an element of \textit{entries}, do
   a. If \textit{p}.[[\text{Key}]] is not empty and \texttt{SameValueZero}(\textit{p}.[[\text{Key}]], \textit{key}) is \texttt{true}, then
      i. Set \textit{p}.[[\text{Value}]] to \textit{value}.
      ii. Return \textit{M}.
   b. If \textit{key} is \texttt{-0}, let \textit{key} be \texttt{+0}.
6. If \textit{key} is \texttt{-0}, let \textit{key} be \texttt{+0}.
7. Let \textit{p} be the \texttt{Record} \{[[\text{Key}]]: \textit{key}, [[\text{Value}]]: \textit{value}\}.
8. Append \textit{p} as the last element of \textit{entries}.
9. Return \textit{M}.

\subsection*{23.1.3.10 get Map.prototype.size}

Map.prototype.size is an accessor property whose set accessor function is \texttt{undefined}. Its get accessor function performs the following steps:

1. Let \textit{M} be the \texttt{this} value.
2. If \texttt{Type}(\textit{M}) is not \texttt{Object}, throw a \texttt{TypeError} exception.
3. If \textit{M} does not have a [[\text{MapData}]] internal slot, throw a \texttt{TypeError} exception.
4. Let \textit{entries} be the \texttt{List} that is \textit{M}.[[\text{MapData}]].
5. Let \texttt{count} be \texttt{0}.
6. For each \texttt{Record} \{[[\text{Key}]], [[\text{Value}]]\} \textit{p} that is an element of \textit{entries}, do
   a. If \textit{p}.[[\text{Key}]] is not empty, set \texttt{count} to \texttt{count}+1.
7. Return \texttt{count}. 
23.1.3.11 Map.prototype.values ()

The following steps are taken:

1. Let \( M \) be the `this` value.
2. Return \(? CreateMapIterator( M, "value"\)).

23.1.3.12 Map.prototype [ @@iterator ] ()

The initial value of the `@@iterator` property is the same function object as the initial value of the `entries` property.

23.1.3.13 Map.prototype [ @@toStringTag ]

The initial value of the `@@toStringTag` property is the String value "Map".

This property has the attributes { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `true` }.

23.1.4 Properties of Map Instances

Map instances are ordinary objects that inherit properties from the Map prototype. Map instances also have a `[[MapData]]` internal slot.

23.1.5 Map Iterator Objects

A Map Iterator is an object, that represents a specific iteration over some specific Map instance object. There is not a named constructor for Map Iterator objects. Instead, map iterator objects are created by calling certain methods of Map instance objects.

23.1.5.1 CreateMapIterator ( `map`, `kind` )

Several methods of Map objects return Iterator objects. The abstract operation CreateMapIterator with arguments `map` and `kind` is used to create such iterator objects. It performs the following steps:

1. If `Type(map)` is not Object, throw a `TypeError` exception.
2. If `map` does not have a `[[MapData]]` internal slot, throw a `TypeError` exception.
3. Let `iterator` be `ObjectCreate(%MapIteratorPrototype%, « [[Map]], [[MapNextIndex]],
   [[MapIterationKind]] »).
4. Set `iterator.``[[Map]]` to `map`.
5. Set `iterator.``[[MapNextIndex]]` to 0.
6. Set `iterator.``[[MapIterationKind]]` to `kind`.
7. Return `iterator`.
23.1.5.2 The %MapIteratorPrototype% Object

All Map Iterator Objects inherit properties from the %MapIteratorPrototype% intrinsic object. The %MapIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %IteratorPrototype% intrinsic object. In addition, %MapIteratorPrototype% has the following properties:

23.1.5.2.1 %MapIteratorPrototype%.next ()

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have all of the internal slots of a Map Iterator Instance (23.1.5.3), throw a TypeError exception.
4. Let m be O.[[Map]].
5. Let index be O.[[MapNextIndex]].
6. Let itemKind be O.[[MapIterationKind]].
7. If m is undefined, return CreateIterResultObject(undefined, true).
8. Assert: m has a [[MapData]] internal slot.
9. Let entries be the List that is m.[[MapData]].
10. Repeat, while index is less than the total number of elements of entries. The number of elements must be redetermined each time this method is evaluated.
   a. Let e be the Record {[[Key]], [[Value]]} that is the value of entries[index].
   b. Set index to index+1.
   c. Set O.[[MapNextIndex]] to index.
   d. If e.[[Key]] is not empty, then
      i. If itemKind is "key", let result be e.[[Key]].
      ii. Else if itemKind is "value", let result be e.[[Value]].
      iii. Else,
         1. Assert: itemKind is "key+value".
         2. Let result be CreateArrayFromList("e.[[Key]], e.[[Value]]").
      iv. Return CreateIterResultObject(result, false).
11. Set O.[[Map]] to undefined.
12. Return CreateIterResultObject(undefined, true).

23.1.5.2.2 %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 }.

23.1.5.3 Properties of Map Iterator Instances
Map Iterator instances are ordinary objects that inherit properties from the `%MapIteratorPrototype%` intrinsic object. Map Iterator instances are initially created with the internal slots described in Table 51.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Map]]</code></td>
<td>The Map object that is being iterated.</td>
</tr>
<tr>
<td><code>[[MapNextIndex]]</code></td>
<td>The integer index of the next Map data element to be examined by this iterator.</td>
</tr>
<tr>
<td><code>[[MapIterationKind]]</code></td>
<td>A String value that identifies what is to be returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;.</td>
</tr>
</tbody>
</table>

### 23.2 Set Objects

Set objects are collections of ECMAScript language values. A distinct value may only occur once as an element of a Set’s collection. Distinct values are discriminated using the `SameValueZero` comparison algorithm.

Set objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Set objects specification is only intended to describe the required observable semantics of Set objects. It is not intended to be a viable implementation model.

#### 23.2.1 The Set Constructor

The Set constructor is the `%Set%` intrinsic object and the initial value of the `Set` property of the global object. When called as a constructor it creates and initializes a new Set object. `Set` is not intended to be called as a function and will throw an exception when called in that manner.

The `Set` constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `Set` behaviour must include a `super` call to the `Set` constructor to create and initialize the subclass instance with the internal state necessary to support the `Set.prototype` built-in methods.

#### 23.2.1.1 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`, `%SetPrototype%`, «`[[SetData]]`»).
3. Set `set.[[SetData]]` to a new empty `List`.
4. If `iterable` is not present, let `iterable` be `undefined`.
5. If `iterable` is either `undefined` or `null`, let `iter` be `undefined`.
6. Else,
   a. Let `adder` be `Get(set, "add")`.
   b. If `IsCallable(adder)` is `false`, throw a `TypeError` exception.
   c. Let `iter` be `GetIterator(iterable)`.
7. If `iter` is `undefined`, return `set`.
8. Repeat,
   a. Let `next` be `? IteratorStep(iter)`.
   b. If `next` is `false`, return `set`.
   c. Let `nextValue` be `? IteratorValue(next)`.
   d. Let `status` be `Call(adder, set, « nextValue.[[Value]] »)`.
   e. If `status` is an `abrupt completion`, return `? IteratorClose(iter, status)`.

### 23.2.2 Properties of the Set Constructor

The value of the `[[Prototype]]` internal slot of the Set constructor is the intrinsic object `%FunctionPrototype%`.

The Set constructor has the following properties:

#### 23.2.2.1 Set.prototype

The initial value of `Set.prototype` is the intrinsic `%SetPrototype%` object.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

#### 23.2.2.2 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.

### 23.2.3 Properties of the Set Prototype Object
The Set prototype object is the intrinsic object `%SetPrototype%`. The value of the [[Prototype]] internal slot of the Set prototype object is the intrinsic object `%ObjectPrototype%`. The Set prototype object is an ordinary object. It does not have a [[SetData]] internal slot.

### 23.2.3.1 `Set.prototype.add (value)`

The following steps are taken:

1. Let `S` be the this value.
2. If `Type(S)` is not `Object`, throw a `TypeError` exception.
3. If `S` does not have a `[[SetData]]` internal slot, throw a `TypeError` exception.
4. Let `entries` be the `List` that is `S.[[SetData]].`
5. For each `e` that is an element of `entries`, do
   a. If `e` is not empty and `SameValueZero(e, value)` is true, then
      i. Return `S`.
6. If `value` is `-0`, let `value` be `+0`.
7. Append `value` as the last element of `entries`.
8. Return `S`.

### 23.2.3.2 `Set.prototype.clear ()`

The following steps are taken:

1. Let `S` be the this value.
2. If `Type(S)` is not `Object`, throw a `TypeError` exception.
3. If `S` does not have a `[[SetData]]` internal slot, throw a `TypeError` exception.
4. Let `entries` be the `List` that is `S.[[SetData]].`
5. For each `e` that is an element of `entries`, do
   a. Replace the element of `entries` whose value is `e` with an element whose value is `empty`.
6. Return `undefined`.

**NOTE** The existing `[[SetData]]` `List` is preserved because there may be existing Set Iterator objects that are suspended midway through iterating over that `List`.

### 23.2.3.3 `Set.prototype.constructor`

The initial value of `Set.prototype.constructor` is the intrinsic object `%Set%`.

### 23.2.3.4 `Set.prototype.delete (value)`

The following steps are taken:
1. Let \( S \) be the \textbf{this} value.
2. If \( \text{Type}(S) \) is not Object, throw a \textbf{TypeError} exception.
3. If \( S \) does not have a \([[\text{SetData}]]) internal slot, throw a \textbf{TypeError} exception.
4. Let \( \textit{entries} \) be the \textbf{List} that is \( S[[\text{SetData}]] \).
5. For each \( e \) that is an element of \( \textit{entries} \), do
   a. If \( e \) is not \textbf{empty} and \( \text{SameValueZero}(e, \textit{value}) \) is \textbf{true}, then
      i. Replace the element of \( \textit{entries} \) whose value is \( e \) with an element whose value is \textbf{empty}.
      ii. Return \textbf{true}.
6. Return \textbf{false}.

\textbf{NOTE} The value \textbf{empty} is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.2.3.5 \textbf{Set.prototype.entries} ( )

The following steps are taken:

1. Let \( S \) be the \textbf{this} value.
2. Return \( \text{? CreateSetIterator}(S, "\text{key+value}")) \).

\textbf{NOTE} For iteration purposes, a Set appears similar to a Map where each entry has the same value for its key and value.

23.2.3.6 \textbf{Set.prototype.forEach} ( \textbf{callbackfn} [ , \textbf{thisArg} ] )

When the \textbf{forEach} method is called with one or two arguments, the following steps are taken:

1. Let \( S \) be the \textbf{this} value.
2. If \( \text{Type}(S) \) is not Object, throw a \textbf{TypeError} exception.
3. If \( S \) does not have a \([[\text{SetData}]) internal slot, throw a \textbf{TypeError} exception.
4. If \( \text{IsCallable}(\text{callbackfn}) \) is \textbf{false}, throw a \textbf{TypeError} exception.
5. If \( \text{thisArg} \) was supplied, let \( T \) be \( \text{thisArg} \); else let \( T \) be \textbf{undefined}.
6. Let \( \textit{entries} \) be the \textbf{List} that is \( S[[\text{SetData}]] \).
7. For each \( e \) that is an element of \( \textit{entries} \), in original insertion order, do
   a. If \( e \) is not \textbf{empty}, then
      i. Perform \( \text{? Call}(\text{callbackfn}, T, \{ \text{e, e, S} \}) \).
8. Return \textbf{undefined}.

\textbf{NOTE} \( \text{callbackfn} \) should be a function that accepts three arguments. \textbf{forEach} calls \( \text{callbackfn} \) once for each value present in the set object, in value insertion order. \( \text{callbackfn} \) 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.

### 23.2.3.7 Set.prototype.has (value)

The following steps are taken:

1. Let `S` be the `this` value.
2. If `Type(S)` is not Object, throw a `TypeError` exception.
3. If `S` does not have a `[[SetData]]` internal slot, throw a `TypeError` exception.
4. Let `entries` be the List that is `S.[[SetData]]`.
5. For each `e` that is an element of `entries`, do
   a. If `e` is not empty and `SameValueZero(e, value)` is `true`, return `true`.
6. Return `false`.

### 23.2.3.8 Set.prototype.keys ()

The initial value of the `keys` property is the same function object as the initial value of the `values` property.

**NOTE**

For iteration purposes, a Set appears similar to a Map where each entry has the same value for its key and value.
23.2.3.9  get Set.prototype.size

Set.prototype.size is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let $S$ be the this value.
2. If Type($S$) is not Object, throw a TypeError exception.
3. If $S$ does not have a [[SetData]] internal slot, throw a TypeError exception.
4. Let entries be the List that is $S$.[[SetData]].
5. Let count be 0.
6. For each $e$ that is an element of entries, do
   a. If $e$ is not empty, set count to count+1.
7. Return count.

23.2.3.10  Set.prototype.values ( )

The following steps are taken:

1. Let $S$ be the this value.
2. Return ? CreateSetIterator($S$, "value").

23.2.3.11  Set.prototype [ @@iterator ] ( )

The initial value of the @@iterator property is the same function object as the initial value of the values property.

23.2.3.12  Set.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "Set".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

23.2.4  Properties of Set Instances

Set instances are ordinary objects that inherit properties from the Set prototype. Set instances also have a [[SetData]] internal slot.

23.2.5  Set Iterator Objects

A Set Iterator is an ordinary object, with the structure defined below, that represents a specific iteration over some specific Set instance object. There is not a named constructor for Set Iterator objects. Instead, set iterator objects are created by calling certain methods of Set instance objects.
23.2.5.1 CreateSetIterator (set, kind)

Several methods of Set objects return Iterator objects. The abstract operation CreateSetIterator with arguments set and kind is used to create such iterator objects. It performs the following steps:

1. If Type(set) is not Object, throw a TypeError exception.
2. If set does not have a [[SetData]] internal slot, throw a TypeError exception.
3. Let iterator be ObjectCreate(%SetIteratorPrototype%, « [[IteratedSet]], [[SetNextIndex]],
   [[SetIterationKind]] »).
4. Set iterator.[[IteratedSet]] to set.
5. Set iterator.[[SetNextIndex]] to 0.
6. Set iterator.[[SetIterationKind]] to kind.
7. Return iterator.

23.2.5.2 The %SetIteratorPrototype% Object

All Set Iterator Objects inherit properties from the %SetIteratorPrototype% intrinsic object. The %SetIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %IteratorPrototype% intrinsic object. In addition, %SetIteratorPrototype% has the following properties:

23.2.5.2.1 %SetIteratorPrototype%.next ()

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have all of the internal slots of a Set Iterator Instance (23.2.5.3), throw a TypeError exception.
4. Let s be O.[[IteratedSet]].
5. Let index be O.[[SetNextIndex]].
6. Let itemKind be O.[[SetIterationKind]].
7. If s is undefined, return CreateIteratorResultObject(undefined, true).
8. Assert: s has a [[SetData]] internal slot.
9. Let entries be the List that is s.[[SetData]].
10. Repeat, while index is less than the total number of elements of entries. The number of elements must be redetermined each time this method is evaluated.
   a. Let e be entries[index].
   b. Set index to index+1.
   c. Set O.[[SetNextIndex]] to index.
   d. If e is not empty, then
      i. If itemKind is "key+value", then
         1. Return CreateIteratorResultObject(CreateArrayFromList(« e, e »), false).
      ii. Return CreateIteratorResultObject(e, false).
11. Set O.[[IteratedSet]] to undefined.
12. Return `CreateIterResultObject(undefined, true)`.

### 23.2.5.2.2 `%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}`.

### 23.2.5.3 Properties of Set Iterator Instances

Set Iterator instances are ordinary objects that inherit properties from the `%SetIteratorPrototype%` intrinsic object. Set Iterator instances are initially created with the internal slots specified in Table 52.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[IteratedSet]]</code></td>
<td>The Set object that is being iterated.</td>
</tr>
<tr>
<td><code>[[SetNextIndex]]</code></td>
<td>The integer index of the next Set data element to be examined by this iterator</td>
</tr>
<tr>
<td><code>[[SetIterationKind]]</code></td>
<td>A String value that identifies what is to be returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;. &quot;key&quot; and &quot;value&quot; have the same meaning.</td>
</tr>
</tbody>
</table>

### 23.3 WeakMap Objects

WeakMap objects are collections of key/value pairs where the keys are objects and values may be arbitrary ECMAScript language values. A WeakMap may be queried to see if it contains a key/value pair with a specific key, but no mechanism is provided for enumerating the objects it holds as keys. If an object that is being used as the key of a WeakMap key/value pair is only reachable by following a chain of references that start within that WeakMap, then that key/value pair is inaccessible and is automatically removed from the WeakMap. WeakMap implementations must detect and remove such key/value pairs and any associated resources.

An implementation may impose an arbitrarily determined latency between the time a key/value pair of a WeakMap becomes inaccessible and the time when the key/value pair is removed from the WeakMap. If this latency was observable to ECMAScript program, it would be a source of indeterminacy that could impact program execution. For that reason, an ECMAScript implementation must not provide any means to observe a key of a WeakMap that does not require the observer to present the observed key.

WeakMap objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of key/value pairs in the collection. The data
structure used in this WeakMap objects specification are only intended to describe the required observable semantics of WeakMap objects. It is not intended to be a viable implementation model.

NOTE WeakMap and WeakSets are intended to provide mechanisms for dynamically associating state with an object in a manner that does not “leak” memory resources if, in the absence of the WeakMap or WeakSet, the object otherwise became inaccessible and subject to resource reclamation by the implementation’s garbage collection mechanisms. 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:


### 23.3.1 TheWeakMap Constructor

The WeakMap constructor is the `%WeakMap%` intrinsic object and the initial value of the WeakMap property of the global object. When called as a constructor it creates and initializes a new WeakMap object. WeakMap is not intended to be called as a function and will throw an exception when called in that manner.

The WeakMap constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified WeakMap behaviour must include a `super` call to the WeakMap constructor to create and initialize the subclass instance with the internal state necessary to support the WeakMap.prototype built-in methods.

#### 23.3.1.1 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, "%WeakMapPrototype%", « [[WeakMapData]] »).
3. Set `map.[[WeakMapData]]` to a new empty List.
4. If `iterable` is not present, let `iterable` be `undefined`. 
5. If `iterable` is either `undefined` or `null`, let `iter` be `undefined`.

6. Else,
   a. Let `adder` be `Get(map, "set")`.
   b. If `IsCallable(adder)` is `false`, throw a `TypeError` exception.
   c. Let `iter` be `GetIterator(iterable)`.

7. If `iter` is `undefined`, return `map`.

8. Repeat,
   a. Let `next` be `? IteratorStep(iter)`.
   b. If `next` is `false`, return `map`.
   c. Let `nextItem` be `? IteratorValue(next)`.
   d. If `Type(nextItem)` is not Object, then
      i. Let `error` be `Completion{[[Type]]: throw, [[Value]]: a newly created TypeError object, [[Target]]: empty}`.
      ii. Return `? IteratorClose(iter, error)`.
   e. Let `k` be `Get(nextItem, "\0")`.
   f. If `k` is an abrupt completion, return `? IteratorClose(iter, k)`.
   g. Let `v` be `Get(nextItem, "\1")`.
   h. If `v` is an abrupt completion, return `? IteratorClose(iter, v)`.
   i. Let `status` be `Call(adder, map, « k.[[Value]], v.[[Value]] »)`.
   j. If `status` is an abrupt completion, return `? IteratorClose(iter, status)`.

**NOTE**
If the parameter `iterable` is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces a two element array-like object whose first element is a value that will be used as a WeakMap key and whose second element is the value to associate with that key.

### 23.3.2 Properties of the WeakMap Constructor

The value of the `[[Prototype]]` internal slot of the WeakMap constructor is the intrinsic object `%FunctionPrototype%`.

The WeakMap constructor has the following properties:

#### 23.3.2.1 WeakMap.prototype

The initial value of `WeakMap.prototype` is the intrinsic object `%WeakMapPrototype%`.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 23.3.3 Properties of the WeakMap Prototype Object
The WeakMap prototype object is the intrinsic object %WeakMapPrototype%. The value of the [[Prototype]] internal slot of the WeakMap prototype object is the intrinsic object %ObjectPrototype%. The WeakMap prototype object is an ordinary object. It does not have a [[WeakMapData]] internal slot.

23.3.3.1 WeakMap.prototype.constructor

The initial value of weakMap.prototype.constructor is the intrinsic object %WeakMap%.

23.3.3.2 WeakMap.prototype.delete (key)

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, throw a TypeError exception.
3. If M does not have a [[WeakMapData]] internal slot, throw a TypeError exception.
4. Let entries be the List that is M.[[WeakMapData]].
5. If Type(key) is not Object, return false.
6. For each Record {[[Key]], [[Value]]} p that is an element of entries, do
   a. If p.[[Key]] is not empty and SameValue(p.[[Key]], key) is true, then
      i. Set p.[[Key]] to empty.
      ii. Set p.[[Value]] to empty.
      iii. Return true.
7. Return false.

NOTE The value empty is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.3.3.3 WeakMap.prototype.get (key)

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, throw a TypeError exception.
3. If M does not have a [[WeakMapData]] internal slot, throw a TypeError exception.
4. Let entries be the List that is M.[[WeakMapData]].
5. If Type(key) is not Object, return undefined.
6. For each Record {[[Key]], [[Value]]} p that is an element of entries, do
   a. If p.[[Key]] is not empty and SameValue(p.[[Key]], key) is true, return p.[[Value]].
7. Return undefined.
23.3.3.4 WeakMap.prototype.has ( key )

The following steps are taken:

1. Let \( M \) be the this value.
2. If \( \text{Type}(M) \) is not Object, throw a TypeError exception.
3. If \( M \) does not have a [[WeakMapData]] internal slot, throw a TypeError exception.
4. Let \( \text{entries} \) be the List that is \( M.[[WeakMapData]] \).
5. If \( \text{Type}(\text{key}) \) is not Object, return false.
6. For each Record \{[[Key]], [[Value]]\} \( p \) that is an element of \( \text{entries} \), do
   a. If \( p.[[Key]] \) is not empty and \( \text{SameValue}(p.[[Key]], \text{key}) \) is true, return true.
7. Return false.

23.3.3.5 WeakMap.prototype.set ( key, value )

The following steps are taken:

1. Let \( M \) be the this value.
2. If \( \text{Type}(M) \) is not Object, throw a TypeError exception.
3. If \( M \) does not have a [[WeakMapData]] internal slot, throw a TypeError exception.
4. Let \( \text{entries} \) be the List that is \( M.[[WeakMapData]] \).
5. If \( \text{Type}(\text{key}) \) is not Object, throw a TypeError exception.
6. For each Record \{[[Key]], [[Value]]\} \( p \) that is an element of \( \text{entries} \), do
   a. If \( p.[[Key]] \) is not empty and \( \text{SameValue}(p.[[Key]], \text{key}) \) is true, then
      i. Set \( p.[[Value]] \) to \( \text{value} \).
      ii. Return \( M \).
7. Let \( p \) be the Record \{[[Key]]: \text{key}, [[Value]]: \text{value} \}.
8. Append \( p \) as the last element of \( \text{entries} \).
9. Return \( M \).

23.3.3.6 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 \}.

23.3.4 Properties of WeakMap Instances

WeakMap instances are ordinary objects that inherit properties from the WeakMap prototype. WeakMap instances also have a [[WeakMapData]] internal slot.
23.4 WeakSet Objects

WeakSet objects are collections of objects. A distinct object may only occur once as an element of a WeakSet’s collection. A WeakSet may be queried to see if it contains a specific object, but no mechanism is provided for enumerating the objects it holds. If an object that is contained by a WeakSet is only reachable by following a chain of references that start within that WeakSet, then that object is inaccessible and is automatically removed from the WeakSet. WeakSet implementations must detect and remove such objects and any associated resources.

An implementation may impose an arbitrarily determined latency between the time an object contained in a WeakSet becomes inaccessible and the time when the object is removed from the WeakSet. If this latency was observable to ECMAScript program, it would be a source of indeterminacy that could impact program execution. For that reason, an ECMAScript implementation must not provide any means to determine if a WeakSet contains a particular object that does not require the observer to present the observed object.

WeakSet objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structure used in this WeakSet objects specification is only intended to describe the required observable semantics of WeakSet objects. It is not intended to be a viable implementation model.

NOTE See the NOTE in 23.3.

23.4.1 The WeakSet Constructor

The WeakSet constructor is the %WeakSet% intrinsic object and the initial value of the WeakSet property of the global object. When called as a constructor it creates and initializes a new WeakSet object. WeakSet is not intended to be called as a function and will throw an exception when called in that manner.

The WeakSet constructor is designed to be subclassable. It may be used as the value in an extends clause of a class definition. Subclass constructors that intend to inherit the specified WeakSet behaviour must include a super call to the WeakSet constructor to create and initialize the subclass instance with the internal state necessary to support the WeakSet.prototype built-in methods.

23.4.1.1 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, "%WeakSetPrototype%", « [[WeakSetData]] »).
3. Set set.[[WeakSetData]] to a new empty List.
4. If iterable is not present, let iterable be undefined.
5. If `iterable` is either `undefined` or `null`, let `iter` be `undefined`.

6. Else,
   a. Let `adder` be `? Get(set, "add")`.
   b. If `IsCallable(adder)` is `false`, throw a `TypeError` exception.
   c. Let `iter` be `GetIterator(iterable)`.

7. If `iter` is `undefined`, return `set`.

8. Repeat,
   a. Let `next` be `? IteratorStep(iter)`.
   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(iter, status)`.

### 23.4.2 Properties of the WeakSet Constructor

The value of the `[[Prototype]]` internal slot of the WeakSet constructor is the intrinsic object `%FunctionPrototype%`.

The WeakSet constructor has the following properties:

#### 23.4.2.1 WeakSet.prototype

The initial value of `WeakSet.prototype` is the intrinsic `%WeakSetPrototype%` object.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 23.4.3 Properties of the WeakSet Prototype Object

The WeakSet prototype object is the intrinsic object `%WeakSetPrototype%`. The value of the `[[Prototype]]` internal slot of the WeakSet prototype object is the intrinsic object `%ObjectPrototype%`. The WeakSet prototype object is an ordinary object. It does not have a `[[WeakSetData]]` internal slot.

#### 23.4.3.1 WeakSet.prototype.add (value)

The following steps are taken:

1. Let `S` be the `this` value.
2. If `Type(S)` is not `Object`, throw a `TypeError` exception.
3. If `S` does not have a `[[WeakSetData]]` internal slot, throw a `TypeError` exception.
4. If `Type(value)` is not `Object`, throw a `TypeError` exception.
5. Let `entries` be the `List` that is `S.[[WeakSetData]]`. 
6. For each \( e \) that is an element of \( \text{entries} \), do
   a. If \( e \) is not empty and \( \text{SameValue}(e, \text{value}) \) is \textbf{true}, then
      i. Return \( S \).
7. Append \( \text{value} \) as the last element of \( \text{entries} \).
8. Return \( S \).

23.4.3.2 \texttt{WeakSet.prototype.constructor}

The initial value of \texttt{WeakSet.prototype.constructor} is the \%WeakSet\% intrinsic object.

23.4.3.3 \texttt{WeakSet.prototype.delete ( value )}

The following steps are taken:

1. Let \( S \) be the \texttt{this} value.
2. If \texttt{Type}(\( S \)) is not Object, throw a \texttt{TypeError} exception.
3. If \( S \) does not have a \texttt{[[WeakSetData]]} internal slot, throw a \texttt{TypeError} exception.
4. If \texttt{Type}(\( \text{value} \)) is not Object, return \textbf{false}.
5. Let \( \text{entries} \) be the \texttt{List} that is \( S.[[\text{WeakSetData}]] \).
6. For each \( e \) that is an element of \( \text{entries} \), do
      a. If \( e \) is not empty and \( \text{SameValue}(e, \text{value}) \) is \textbf{true}, then
         i. Replace the element of \( \text{entries} \) whose value is \( e \) with an element whose value is \textbf{empty}.
         ii. Return \textbf{true}.
5. Return \textbf{false}.

\textbf{NOTE} The value \textbf{empty} is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.4.3.4 \texttt{WeakSet.prototype.has ( value )}

The following steps are taken:

1. Let \( S \) be the \texttt{this} value.
2. If \texttt{Type}(\( S \)) is not Object, throw a \texttt{TypeError} exception.
3. If \( S \) does not have a \texttt{[[WeakSetData]]} internal slot, throw a \texttt{TypeError} exception.
4. Let \( \text{entries} \) be the \texttt{List} that is \( S.[[\text{WeakSetData}]] \).
5. If \texttt{Type}(\( \text{value} \)) is not Object, return \textbf{false}.
6. For each \( e \) that is an element of \( \text{entries} \), do
      a. If \( e \) is not empty and \( \text{SameValue}(e, \text{value}) \) is \textbf{true}, return \textbf{true}.
7. Return \textbf{false}.
23.4.3.5 WeakSet.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "WeakSet".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

23.4.4 Properties of WeakSet Instances

WeakSet instances are ordinary objects that inherit properties from the WeakSet prototype. WeakSet instances also have a [[WeakSetData]] internal slot.

24 Structured Data

24.1 ArrayBuffer Objects

24.1.1 Abstract Operations For ArrayBuffer Objects

24.1.1.1 AllocateArrayBuffer ( constructor, byteLength )

The abstract operation AllocateArrayBuffer with arguments constructor and byteLength is used to create an ArrayBuffer object. It performs the following steps:

1. Let obj be ? OrdinaryCreateFromConstructor(constructor, "%ArrayBufferPrototype", « [ArrayBufferData], [ArrayBufferByteLength] »).
2. Assert: byteLength is an integer value ≥ 0.
3. Let block be ? CreateByteDataBlock(byteLength).
4. Set obj.[ArrayBufferData] to block.
5. Set obj.[ArrayBufferByteLength] to byteLength.
6. Return obj.

24.1.1.2 IsDetachedBuffer ( arrayBuffer )

The abstract operation IsDetachedBuffer with argument arrayBuffer performs the following steps:

1. Assert: Type(arrayBuffer) is Object and it has an [[ArrayBufferData]] internal slot.
2. If arrayBuffer.[ArrayBufferData] is null, return true.
3. Return false.
24.1.1.3 DetachArrayBuffer (arrayBuffer)

The abstract operation DetachArrayBuffer with argument arrayBuffer performs the following steps:

1. Assert: Type(arrayBuffer) is Object and it has [[ArrayBufferData]] and [[ArrayBufferByteLength]] internal slots.
2. Assert: IsSharedArrayBuffer(arrayBuffer) is false.
3. Set arrayBuffer.([[ArrayBufferData]]) to null.
4. Set arrayBuffer.([[ArrayBufferByteLength]]) to 0.
5. 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 implementation or host environment may define such operations.

24.1.1.4 CloneArrayBuffer (srcBuffer, srcByteOffset, srcLength [, cloneConstructor ])

The abstract operation CloneArrayBuffer takes four parameters, an ArrayBuffer srcBuffer, an integer offset srcByteOffset, an integer length srcLength, and optionally a constructor function cloneConstructor. 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. This operation performs the following steps:

1. Assert: Type(srcBuffer) is Object and it has an [[ArrayBufferData]] internal slot.
2. If cloneConstructor is not present, then
   a. Set cloneConstructor to ? SpeciesConstructor(srcBuffer, %ArrayBuffer%).
   b. If IsDetachedBuffer(srcBuffer) is true, throw a TypeError exception.
3. Else, Assert: IsConstructor(cloneConstructor) is true.
4. Let srcBlock be srcBuffer.([[ArrayBufferData]])
5. Let targetBuffer be ? AllocateArrayBuffer(cloneConstructor, srcLength).
6. If IsDetachedBuffer(srcBuffer) is true, throw a TypeError exception.
7. Let targetBlock be targetBuffer.([[ArrayBufferData]])
8. Perform CopyDataBlockBytes(targetBlock, 0, srcBlock, srcByteOffset, srcLength).

24.1.1.5 RawBytesToNumber (type, rawBytes, isLittleEndian)

The abstract operation RawBytesToNumber takes three parameters, a String type, a List rawBytes, and a Boolean isLittleEndian. This operation performs the following steps:

1. Let elementSize be the Number value of the Element Size value specified in Table 50 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 754-2008 binary32 value.
   b. If `value` is an IEEE 754-2008 binary32 NaN value, return the `NaN` Number value.
   c. Return the Number value that corresponds to `value`.

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 754-2008 binary64 value.
   b. If `value` is an IEEE 754-2008 binary64 NaN value, return the `NaN` Number value.
   c. Return the Number value that corresponds to `value`.

5. If the first code unit of `type` is "U", then
   a. Let `intValue` be the byte elements of `rawBytes` concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.

6. Else,
   a. Let `intValue` be the byte elements of `rawBytes` concatenated and interpreted as a bit string encoding of a binary little-endian 2’s complement number of bit length `elementTypeSize × 8`.

7. Return the Number value that corresponds to `intValue`.

24.1.1.6 GetValueFromBuffer ( `arrayBuffer`, `byteIndex`, `type`, `isTypedArray`, `order` [ , `isLittleEndian` ] )

The abstract operation GetValueFromBuffer takes six parameters, an ArrayBuffer or SharedArrayBuffer `arrayBuffer`, an integer `byteIndex`, a String `type`, a Boolean `isTypedArray`, a String `order`, and optionally a Boolean `isLittleEndian`. This operation performs the following steps:

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: `byteIndex` is an integer value ≥ 0.
4. Let `block` be `arrayBuffer.[[ArrayBufferData]]`.
5. Let `elementTypeSize` be the Number value of the Element Size value specified in Table 50 for Element Type `type`.
6. 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`. `[[EventLists]]` whose `[[AgentSignifier]]` is `AgentSignifier()`.
   c. If `isTypedArray` is `true` and `type` is "Int8", "Uint8", "Int16", "Uint16", "Int32", or "Uint32", let `noTear` be `true`; otherwise let `noTear` be `false`.
   d. Let `rawValue` be a List of length `elementTypeSize` of nondeterministically chosen byte values.
   e. NOTE: In implementations, `rawValue` is the result of a non-atomic or atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the memory model
to describe observable behaviour of hardware with weak consistency.

f. Let readEvent be ReadSharedMemory( [[Order]]: order, [[NoTear]]: noTear, [[Block]]: block, [[BytIndex]]: bytIndex, [[ElementSize]]: elementSize ).

g. Append readEvent to eventList.

h. Append Chosen Value Record { [[Event]]: readEvent, [[ChosenValue]]: rawValue } to execution. [[ChosenValues]].

7. Else, let rawValue be a List of elementSize containing, in order, the elementSize sequence of bytes starting with block[bytIndex].

8. If isLittleEndian is not present, set isLittleEndian to the value of the [[LittleEndian]] field of the surrounding agent’s Agent Record.

9. Return RawBytesToNumber(type, rawValue, isLittleEndian).

24.1.1.7 NumberToRawBytes( type, value, isLittleEndian )

The abstract operation NumberToRawBytes takes three parameters, a String type, a Number value, and a Boolean isLittleEndian. This operation performs the following steps:

1. If type is "Float32", then
   a. Let rawBytes be a List containing the 4 bytes that are the result of converting value to IEEE 754-2008 binary32 format using “Round to nearest, ties to even” rounding mode. If isLittleEndian is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawBytes may be set to any implementation chosen IEEE 754-2008 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 containing the 8 bytes that are the IEEE 754-2008 binary64 format encoding of value. If isLittleEndian is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawBytes may be set to any implementation chosen IEEE 754-2008 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 Number value of the Element Size specified in Table 50 for Element Type type.
   b. Let convOp be the abstract operation named in the Conversion Operation column in Table 50 for Element Type type.
   c. Let intValue be convOp(value).
   d. If intValue ≥ 0, then
      i. Let rawBytes be a List containing the n-byte binary encoding of intValue. If isLittleEndian is false, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.
   e. Else,
i. Let `rawBytes` be a List containing the \(n\)-byte binary 2's complement encoding of `intValue`. If `isLittleEndian` is `false`, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.

4. Return `rawBytes`.

### 24.1.1.8 SetValueInBuffer (arrayBuffer, byteIndex, type, value, isTypedArray, order [ , isLittleEndian ])

The abstract operation `SetValueInBuffer` takes seven parameters, an ArrayBuffer or SharedArrayBuffer `arrayBuffer`, an integer `byteIndex`, a String `type`, a Number `value`, a Boolean `isTypedArray`, a String `order`, and optionally a Boolean `isLittleEndian`. This operation performs the following steps:

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: `byteIndex` is an integer value \( \geq 0 \).
4. Assert: `Type(value)` is Number.
5. Let `block` be `arrayBuffer`[[ArrayBufferData]].
6. Let `elementSize` be the Number value of the Element Size value specified in Table 50 for Element Type `type`.
7. If `isLittleEndian` is not present, set `isLittleEndian` to the value of the `[[LittleEndian]]` field of the surrounding agent's Agent Record.
8. Let `rawBytes` be `NumberToRawBytes(type, value, isLittleEndian)`.
9. 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` `[[EventLists]]` whose `[[AgentSignifier]]` is `AgentSignifier()`.
   c. If `isTypedArray` is `true` and `type` is `"Int8", "Uint8", "Int16", "Uint16", "Int32",` or `"UInt32"`, 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`.
10. Else, store the individual bytes of `rawBytes` into `block`, in order, starting at `block[byteIndex]`.
11. Return `NormalCompletion(undefined)`.

### 24.1.1.9 GetModifySetValueInBuffer( arrayBuffer, byteIndex, type, value, op [ , isLittleEndian ])

The abstract operation `GetModifySetValueInBuffer` takes six parameters, a SharedArrayBuffer `arrayBuffer`, a nonnegative integer `byteIndex`, a String `type`, a Number `value`, a semantic function `op`, and optionally a Boolean `isLittleEndian`. This operation performs the following steps:

1. Assert: `IsSharedArrayBuffer(arrayBuffer)` is `true`.
2. Assert: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `type`. 

3. Assert: `byteIndex` is an integer value ≥ 0.
4. Assert: `Type(value)` is Number.
5. Let `block` be `arrayBuffer.([[ArrayBufferData]])`.
6. Let `elementSize` be the Number value of the Element Size value specified in Table 50 for Element Type `type`.
7. If `isLittleEndian` is not present, set `isLittleEndian` to the value of the `[[LittleEndian]]` field of the surrounding agent’s Agent Record.
8. Let `rawBytes` be `NumberToRawBytes(type, value, isLittleEndian)`.
9. Let `execution` be the `[[CandidateExecution]]` field of the surrounding agent’s Agent Record.
10. Let `eventList` be the `[[EventList]]` field of the element in `execution. [[EventLists]]` whose `[[AgentSignifier]]` is `AgentSignifier()`.
11. Let `rawBytesRead` be a List of length `elementSize` of nondeterministically chosen byte values.
12. 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.
13. Let `rmwEvent` be `ReadModifyWriteSharedMemory{ [[Order]]: "SeqCst", [[NoTear]]: true, [[Block]]: `block`, [[ByteIndex]]: `byteIndex`, [[ElementSize]]: `elementSize`, [[Payload]]: `rawBytes`, [[ModifyOp]]: `op` }.
15. Append `Chosen Value Record { [[Event]]: `rmwEvent`, [[ChosenValue]]: `rawBytesRead` } to `execution. [[Values]]`.
16. Return `RawBytesToNumber(type, rawBytesRead, isLittleEndian)`.

### 24.1.2 The ArrayBuffer Constructor

The ArrayBuffer constructor is the `%ArrayBuffer%` intrinsic object and the initial value of the `ArrayBuffer` property of the global object. When called as a constructor it creates and initializes a new ArrayBuffer object. `ArrayBuffer` is not intended to be called as a function and will throw an exception when called in that manner.

The `ArrayBuffer` 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 `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.

#### 24.1.2.1 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)`.

### 24.1.3 Properties of the ArrayBuffer Constructor

The value of the `[[Prototype]]` internal slot of the ArrayBuffer constructor is the intrinsic object `%FunctionPrototype%`. The ArrayBuffer constructor has the following properties:

#### 24.1.3.1 ArrayBuffer.isView (arg)

The `isView` function takes one argument `arg`, and performs, the following steps are taken:

1. If `Type(arg)` is not Object, return `false`.
2. If `arg` has a `[[ViewedArrayBuffer]]` internal slot, return `true`.
3. Return `false`.

#### 24.1.3.2 ArrayBuffer.prototype

The initial value of `ArrayBuffer.prototype` is the intrinsic object `%ArrayBufferPrototype%`. This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

#### 24.1.3.3 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` object’s constructor to create a derived object. However, a subclass constructor may over-ride that default behaviour by redefining its `@@species` property.

### 24.1.4 Properties of the ArrayBuffer Prototype Object

The ArrayBuffer prototype object is the intrinsic object `%ArrayBufferPrototype%`. The value of the `[[Prototype]]` internal slot of the ArrayBuffer prototype object is the intrinsic object `%ObjectPrototype%`.
The ArrayBuffer prototype object is an ordinary object. It does not have an [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal slot.

24.1.4.1 get ArrayBuffer.prototype.byteLength

`ArrayBuffer.prototype.byteLength` is an accessor property whose set accessor function is `undefined`. Its get accessor function performs the following steps:

1. Let `O` be the `this` value.
2. If `Type(O)` is not Object, throw a `TypeError` exception.
3. If `O` does not have an [[ArrayBufferData]] internal slot, throw a `TypeError` exception.
4. If `IsSharedArrayBuffer(O)` is `true`, throw a `TypeError` exception.
5. If `IsDetachedBuffer(O)` is `true`, throw a `TypeError` exception.
6. Let `length` be `O.[[ArrayBufferByteLength]].`
7. Return `length`.

24.1.4.2 ArrayBuffer.prototype.constructor

The initial value of `ArrayBuffer.prototype.constructor` is the intrinsic object `%ArrayBuffer%`.

24.1.4.3 ArrayBuffer.prototype.slice ( `start, end` )

The following steps are taken:

1. Let `O` be the `this` value.
2. If `Type(O)` is not Object, throw a `TypeError` exception.
3. If `O` does not have an [[ArrayBufferData]] internal slot, throw a `TypeError` exception.
4. If `IsSharedArrayBuffer(O)` is `true`, throw a `TypeError` exception.
5. If `IsDetachedBuffer(O)` is `true`, throw a `TypeError` exception.
6. Let `len` be `O.[[ArrayBufferByteLength]].`
7. Let `relativeStart` be `? ToInteger(start)`.
8. If `relativeStart < 0`, let `first` be `max((len + relativeStart), 0)`; else let `first` be `min(relativeStart, len)`.
9. If `end` is `undefined`, let `relativeEnd` be `len`; else let `relativeEnd` be `? ToInteger(end)`.
10. If `relativeEnd < 0`, let `final` be `max((len + relativeEnd), 0)`; else let `final` be `min(relativeEnd, len)`.
11. Let `newLen` be `max(final-first, 0)`.
12. Let `ctor` be `? SpeciesConstructor(O, %ArrayBuffer%)`.
14. If `new` does not have an [[ArrayBufferData]] internal slot, throw a `TypeError` exception.
15. If `IsSharedArrayBuffer(new)` is `true`, throw a `TypeError` exception.
16. If `IsDetachedBuffer(new)` is `true`, throw a `TypeError` exception.
17. If `SameValue(new, O)` is `true`, throw a `TypeError` exception.
18. If \( \text{new} \).\([\text{ArrayBufferByteLength}] < \text{newLen}, \) throw a \text{TypeError} exception.
19. NOTE: Side-effects of the above steps may have detached \( O \).
20. If \( \text{IsDetachedBuffer}(O) \) is \text{true}, throw a \text{TypeError} exception.
21. Let \( \text{fromBuf} \) be \( O.\[\text{ArrayBufferData}] \).
22. Let \( \text{toBuf} \) be \( \text{new}.\[\text{ArrayBufferData}] \).
23. Perform \( \text{CopyDataBlockBytes}(\text{toBuf}, 0, \text{fromBuf}, \text{first}, \text{newLen}) \).
24. Return \( \text{new} \).

24.1.4.4 \text{ArrayBuffer.prototype} [\ @@\text{toStringTag} ]

The initial value of the \( @@\text{toStringTag} \) property is the String value "\text{ArrayBuffer}". This property has the attributes \{ \([\text{Writable}]: \text{false}, \([\text{Enumerable}]: \text{false}, \)[\text{Configurable}]: \text{true} \}.

24.1.5 Properties of the \text{ArrayBuffer} Instances

\text{ArrayBuffer} instances inherit properties from the \text{ArrayBuffer prototype} object. \text{ArrayBuffer} instances each have an \([\text{ArrayBufferData}] \) internal slot and an \([\text{ArrayBufferByteLength}] \) internal slot.

\text{ArrayBuffer} instances whose \([\text{ArrayBufferData}] \) is \text{null} are considered to be detached and all operators to access or modify data contained in the \text{ArrayBuffer} instance will fail.

24.2 \text{SharedArrayBuffer} Objects

24.2.1 Abstract Operations for \text{SharedArrayBuffer} Objects

24.2.1.1 \text{AllocateSharedArrayBuffer}( constructor, byteLength )

The abstract operation \text{AllocateSharedArrayBuffer} with arguments \text{constructor} and \text{byteLength} is used to create a \text{SharedArrayBuffer} object. It performs the following steps:

1. Let \( \text{obj} \) be \( \text{OrdinaryCreateFromConstructor}(\text{constructor}, "\%\text{SharedArrayBufferPrototype}\%", \text{«} \[[\text{ArrayBufferData}]], \[[\text{ArrayBufferByteLength}] \] \text{»}) \).
2. Assert: \( \text{byteLength} \) is a nonnegative integer.
3. Let \( \text{block} \) be \( \text{CreateSharedByteDataBlock} \)(\text{byteLength})
4. Set \( \text{obj}.\[[\text{ArrayBufferData}] \] to \( \text{block} \).
5. Set \( \text{obj}.\[[\text{ArrayBufferByteLength}] \] to \( \text{byteLength} \).
6. Return \( \text{obj} \).

24.2.1.2 \text{IsSharedArrayBuffer}( \text{obj} )
IsSharedArrayBuffer tests whether an object is an ArrayBuffer, a SharedArrayBuffer, or a subtype of either. It performs the following steps:

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`.

### 24.2.2 The SharedArrayBuffer Constructor

The SharedArrayBuffer constructor is the `%SharedArrayBuffer%` intrinsic object and the initial value of the `SharedArrayBuffer` property of the `global` object. When called as a constructor it creates and initializes a new SharedArrayBuffer object. `SharedArrayBuffer` is not intended to be called as a function and will throw an exception when called in that manner.

The `SharedArrayBuffer` 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 `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.

**NOTE** Unlike an `ArrayBuffer`, a `SharedArrayBuffer` cannot become detached, and its internal `[[ArrayBufferData]]` slot is never `null`.

### 24.2.2.1 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)`.

### 24.2.3 Properties of the SharedArrayBuffer Constructor

The value of the `[[Prototype]]` internal slot of the SharedArrayBuffer constructor is the intrinsic object `%FunctionPrototype%`.

The SharedArrayBuffer constructor has the following properties:
24.2.3.1 SharedArrayBuffer.prototype

The initial value of SharedArrayBuffer.prototype is the intrinsic object %SharedArrayBufferPrototype%.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

24.2.3.2 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]".

24.2.4 Properties of the SharedArrayBuffer Prototype Object

The SharedArrayBuffer prototype object is the intrinsic object %SharedArrayBufferPrototype%. The value of the [[Prototype]] internal slot of the SharedArrayBuffer prototype object is the intrinsic object %ObjectPrototype%. The SharedArrayBuffer prototype object is an ordinary object. It does not have an [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal slot.

24.2.4.1 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. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
4. If IsSharedArrayBuffer(O) is false, throw a TypeError exception.
5. Let length be O.[[ArrayBufferByteLength]].
6. Return length.

24.2.4.2 SharedArrayBuffer.prototype.constructor

The initial value of SharedArrayBuffer.prototype.constructor is the intrinsic object %SharedArrayBuffer%. 

24.2.4.3 SharedArrayBuffer.prototype.slice( start, end )

The following steps are taken:
1. Let $O$ be the this value.
2. If $\text{Type}(O)$ is not Object, throw a TypeError exception.
3. If $O$ does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
4. If $\text{IsSharedArrayBuffer}(O)$ is false, throw a TypeError exception.
5. Let $len$ be $O$.[[ArrayBufferDataByteLength]].
6. Let $\text{relativeStart}$ be $\text{ToInteger}(\text{start})$.
7. If $\text{relativeStart}$ < 0, let $\text{first}$ be $\text{max}(\text{len} + \text{relativeStart}, 0)$; else let $\text{first}$ be $\text{min}(\text{relativeStart}, \text{len})$.
8. If $\text{end}$ is undefined, let $\text{relativeEnd}$ be $\text{len}$; else let $\text{relativeEnd}$ be $\text{ToInteger}(\text{end})$.
9. If $\text{relativeEnd}$ < 0, let $\text{final}$ be $\text{max}(\text{len} + \text{relativeEnd}, 0)$; else let $\text{final}$ be $\text{min}(\text{relativeEnd}, \text{len})$.
10. Let $\text{newLen}$ be $\text{max}(\text{final} - \text{first}, 0)$.
11. Let $\text{ctor}$ be $\text{SpeciesConstructor}(O, \%\text{SharedArrayBuffer}%)$.
12. Let $\text{new}$ be $\text{Construct}(\text{ctor}, \langle \text{newLen} \rangle)$.
13. If $\text{new}$ does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
14. If $\text{IsSharedArrayBuffer}(\text{new})$ is false, throw a TypeError exception.
15. If $\text{new}.[[\text{ArrayBufferData}]]$ and $O.[[\text{ArrayBufferData}]]$ are the same Shared Data Block values, throw a TypeError exception.
16. If $\text{new}.[[\text{ArrayBufferDataByteLength}]] < \text{newLen}$, throw a TypeError exception.
17. Let $\text{fromBuf}$ be $O.[[\text{ArrayBufferData}]]$.
18. Let $\text{toBuf}$ be $\text{new}.[[\text{ArrayBufferData}]]$.
19. Perform $\text{CopyDataBlockBytes}(\text{toBuf}, 0, \text{fromBuf}, \text{first}, \text{newLen})$.
20. Return $\text{new}$.

### 24.2.4 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 }.

### 24.2.5 Properties of the SharedArrayBuffer Instances

SharedArrayBuffer instances inherit properties from the SharedArrayBuffer prototype object. SharedArrayBuffer instances each have an [[ArrayBufferData]] internal slot and an [[ArrayBufferDataByteLength]] internal slot.

| NOTE | SharedArrayBuffer instances, unlike ArrayBuffer instances, are never detached. |

### 24.3 DataView Objects

#### 24.3.1 Abstract Operations For DataView Objects
24.3.1.1 GetViewValue (view, requestIndex, isLittleEndian, type)

The abstract operation GetViewValue with arguments view, requestIndex, isLittleEndian, and type is used by functions on DataView instances to retrieve values from the view’s buffer. It performs the following steps:

1. If Type(view) is not Object, throw a TypeError exception.
2. If view does not have a [[DataView]] internal slot, throw a TypeError exception.
3. Assert: view has a [[ViewedArrayBuffer]] internal slot.
4. Let getIndex be ? ToIndex(requestIndex).
5. Set isLittleEndian to ToBoolean(isLittleEndian).
6. Let buffer be view.[[ViewedArrayBuffer]].
7. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
8. Let viewOffset be view.[[ByteOffset]].
9. Let viewSize be view.[[ByteLength]].
10. Let elementSize be the Number value of the Element Size value specified in Table 50 for Element Type type.
11. If getIndex + elementSize > viewSize, throw a RangeError exception.
12. Let bufferIndex be getIndex + viewOffset.
13. Return GetValueFromBuffer(buffer, bufferIndex, type, false, "Unordered", isLittleEndian).

24.3.1.2 SetViewValue (view, requestIndex, isLittleEndian, type, value)

The abstract operation SetViewValue with arguments view, requestIndex, isLittleEndian, type, and value is used by functions on DataView instances to store values into the view’s buffer. It performs the following steps:

1. If Type(view) is not Object, throw a TypeError exception.
2. If view does not have a [[DataView]] internal slot, throw a TypeError exception.
3. Assert: view has a [[ViewedArrayBuffer]] internal slot.
4. Let getIndex be ? ToIndex(requestIndex).
5. Let numberValue be ? ToNumber(value).
6. Set isLittleEndian to ToBoolean(isLittleEndian).
7. Let buffer be view.[[ViewedArrayBuffer]].
8. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
9. Let viewOffset be view.[[ByteOffset]].
10. Let viewSize be view.[[ByteLength]].
11. Let elementSize be the Number value of the Element Size value specified in Table 50 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).
24.3.2 The DataView Constructor

The DataView constructor is the `%DataView%` intrinsic object and the initial value of the `DataView` property of the `global object`. When called as a constructor it creates and initializes a new DataView object. `DataView` is not intended to be called as a function and will throw an exception when called in that manner.

The `DataView` 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 `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.

24.3.2.1 DataView ( `buffer [, byteOffset [, byteLength ]]` )

When the `DataView` is called with at least one argument `buffer`, the following steps are taken:

1. If `NewTarget` is `undefined`, throw a `TypeError` exception.
2. If `Type(buffer)` is not `Object`, throw a `TypeError` exception.
3. If `buffer` does not have an `[[ArrayBufferData]]` internal slot, throw a `TypeError` exception.
4. Let `offset` be `? ToIndex(byteOffset)`.
5. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
6. Let `bufferByteLength` be `buffer.[[ArrayBufferByteLength]]`.
7. If `offset > bufferByteLength`, throw a `RangeError` exception.
8. If `byteLength` is either not present or `undefined`, then
   a. Let `viewByteLength` be `bufferByteLength - offset`.
9. Else,
   a. Let `viewByteLength` be `? ToIndex(byteLength)`.
   b. If `offset+viewByteLength > bufferByteLength`, throw a `RangeError` exception.
11. Set `O. [[ViewedArrayBuffer]]` to `buffer`.
12. Set `O. [[ByteLength]]` to `viewByteLength`.
13. Set `O. [[ByteOffset]]` to `offset`.

24.3.3 Properties of the DataView Constructor

The value of the `[[Prototype]]` internal slot of the `DataView` constructor is the intrinsic object `%FunctionPrototype%`.

The DataView constructor has the following properties:
24.3.3.1 DataView.prototype

The initial value of DataView.prototype is the intrinsic object %DataViewPrototype%.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

24.3.4 Properties of the DataView Prototype Object

The DataView prototype object is the intrinsic object %DataViewPrototype%. The value of the [[Prototype]] internal slot of the DataView prototype object is the intrinsic object %ObjectPrototype%. The DataView prototype object is an ordinary object. It does not have a [[DataView]], [[ViewedArrayBuffer]], [[ByteLength]], or [[ByteOffset]] internal slot.

24.3.4.1 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. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[DataView]] internal slot, throw a TypeError exception.
4. Assert: O has a [[ViewedArrayBuffer]] internal slot.
5. Let buffer be O.[[ViewedArrayBuffer]].
6. Return buffer.

24.3.4.2 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. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[DataView]] internal slot, throw a TypeError exception.
4. Assert: O has a [[ViewedArrayBuffer]] internal slot.
5. Let buffer be O.[[ViewedArrayBuffer]].
6. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
7. Let size be O.[[ByteLength]].
8. Return size.

24.3.4.3 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. If `Type(O)` is not `Object`, throw a **TypeError** exception.
3. If \( O \) does not have a `[[DataView]]` internal slot, throw a **TypeError** exception.
4. Assert: \( O \) has a `[[ViewedArrayBuffer]]` internal slot.
5. Let `buffer` be \( O.[[ViewedArrayBuffer]] \).
6. If `IsDetachedBuffer(buffer)` is **true**, throw a **TypeError** exception.
7. Let `offset` be \( O.[[ByteOffset]] \).
8. Return `offset`.

### 24.3.4.4 DataView.prototype.constructor

The initial value of `DataView.prototype.constructor` is the intrinsic object `%DataView%`.

### 24.3.4.5 DataView.prototype.getFloat32 ( **byteOffset**[ , **littleEndian** ] )

When the get**Float32** 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, let `littleEndian` be **false**.

### 24.3.4.6 DataView.prototype.getFloat64 ( **byteOffset**[ , **littleEndian** ] )

When the get**Float64** 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, let `littleEndian` be **false**.
3. Return `? ViewValue(v, byteOffset, littleEndian, "Float64")`.

### 24.3.4.7 DataView.prototype.getInt8 ( **byteOffset** )

When the get**Int8** method is called with argument `byteOffset`, the following steps are taken:

1. Let \( v \) be the **this** value.
2. Return `? ViewValue(v, byteOffset, **true**, "Int8")`.

### 24.3.4.8 DataView.prototype.getInt16 ( **byteOffset**[ , **littleEndian** ] )
When the `getInt16` method is called with argument `byteOffset` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `? GetViewValue(v, byteOffset, littleEndian, "Int16")`.

### 24.3.4.9 DataView.prototype.getInt32 ( `byteOffset` [ , `littleEndian` ] )

When the `getInt32` method is called with argument `byteOffset` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `undefined`.
3. Return `? GetViewValue(v, byteOffset, littleEndian, "Int32")`.

### 24.3.4.10 DataView.prototype.getUint8 ( `byteOffset` )

When the `getUint8` method is called with argument `byteOffset`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. Return `? GetViewValue(v, byteOffset, true, "Uint8")`.

### 24.3.4.11 DataView.prototype.getUint16 ( `byteOffset` [ , `littleEndian` ] )

When the `getUint16` method is called with argument `byteOffset` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `? GetViewValue(v, byteOffset, littleEndian, "Uint16")`.

### 24.3.4.12 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, let `littleEndian` be `false`.

### 24.3.4.13 DataView.prototype.setFloat32 ( `byteOffset`, `value` [ , `littleEndian` ] )
When the `setFloat32` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `SetViewValue(v, byteOffset, littleEndian, "Float32", value)`.

### 24.3.4.14 DataView.prototype.setFloat64 ( `byteOffset`, `value` [, `littleEndian` ] )

When the `setFloat64` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `SetViewValue(v, byteOffset, littleEndian, "Float64", value)`.

### 24.3.4.15 DataView.prototype.setInt8 ( `byteOffset`, `value` )

When the `setInt8` method is called with arguments `byteOffset` and `value`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. Return `SetViewValue(v, byteOffset, true, "Int8", value)`.

### 24.3.4.16 DataView.prototype.setInt16 ( `byteOffset`, `value` [, `littleEndian` ] )

When the `setInt16` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `SetViewValue(v, byteOffset, littleEndian, "Int16", value)`.

### 24.3.4.17 DataView.prototype.setInt32 ( `byteOffset`, `value` [, `littleEndian` ] )

When the `setInt32` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let \( v \) be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.
3. Return `SetViewValue(v, byteOffset, littleEndian, "Int32", value)`.

### 24.3.4.18 DataView.prototype.setUint8 ( `byteOffset`, `value` )
When the `setUint8` method is called with arguments `byteOffset` and `value`, the following steps are taken:

1. Let `v` be the `this` value.
2. Return `? SetViewValue(v, byteOffset, true, "Uint8", value)`.

### 24.3.4.19 DataView.prototype.setUint16 ( `byteOffset`, `value` [ , `littleEndian` ] )

When the `setUint16` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let `v` be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.

### 24.3.4.20 DataView.prototype.setUint32 ( `byteOffset`, `value` [ , `littleEndian` ] )

When the `setUint32` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian`, the following steps are taken:

1. Let `v` be the `this` value.
2. If `littleEndian` is not present, let `littleEndian` be `false`.

### 24.3.4.21 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 }.

### 24.3.5 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.

### 24.4 The Atomics Object
The Atomics object is the %Atomics% intrinsic object and the initial value of the Atomics property of the global object. The Atomics object is a single ordinary object.

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.

The value of the [[Prototype]] internal slot of the Atomics object is the intrinsic object %ObjectPrototype%.

The Atomics object does not have a [[Construct]] internal method; it is not possible to use the Atomics object as a constructor with the new operator.

The Atomics object does not have a [[Call]] internal method; it is not possible to invoke the Atomics object as a function.

NOTE For informative guidelines for programming and implementing shared memory in ECMAScript, please see the notes at the end of the memory model section.

### 24.4.1 Abstract Operations for Atomics

#### 24.4.1.1 ValidateSharedIntegerTypedArray(typedArray [, onlyInt32 ])

The abstract operation ValidateSharedIntegerTypedArray takes one argument typedArray and an optional Boolean onlyInt32. It performs the following steps:

1. If the onlyInt32 argument was not provided, set onlyInt32 to false.
2. If Type(typedArray) is not Object, throw a TypeError exception.
3. If typedArray does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
4. Let typeName be typedArray.[[TypedArrayName]].
5. If onlyInt32 is true, then
   a. If typeName is not "Int32Array", throw a TypeError exception.
6. Else,
   a. If typeName is not "Int8Array", "Uint8Array", "Int16Array", "Uint16Array", "Int32Array", or "Uint32Array", throw a TypeError exception.
7. Assert: typedArray has a [[ViewedArrayBuffer]] internal slot.
8. Let buffer be typedArray.[[ViewedArrayBuffer]].
9. If IsSharedArrayBuffer(buffer) is false, throw a TypeError exception.
10. Return buffer.

#### 24.4.1.2 ValidateAtomicAccess(typedArray, requestIndex )
The abstract operation ValidateAtomicAccess takes two arguments, `typedArray` and `requestIndex`. It performs the following steps:

1. Assert: `typedArray` is an Object that has a `[[ViewedArrayBuffer]]` internal slot.
2. Let `accessIndex` be ? `ToIndex(requestIndex)`.
3. Let `length` be `typedArray.([[ArrayLength]])`.
4. Assert: `accessIndex` ≥ 0.
5. If `accessIndex` ≥ `length`, throw a `RangeError` exception.
6. Return `accessIndex`.

### 24.4.1.3 GetWaiterList( `block`, `i` )

A `WaiterList` is a semantic object that contains an ordered list of those agents that are waiting on a location `(block, i)` in shared memory; `block` is a `Shared Data Block` and `i` a byte offset into the memory of `block`.

The agent cluster has a store of `WaiterList` objects; the store is indexed by `(block, i)`. `WaiterLists` are agent-independent: a lookup in the store of `WaiterLists` by `(block, i)` will result in the same `WaiterList` object in any agent in the agent cluster.

Operations on a `WaiterList` -- adding and removing waiting agents, traversing the list of agents, suspending and waking agents on the list -- may only be performed by agents that have entered the `WaiterList`'s critical section.

The abstract operation GetWaiterList takes two arguments, a `Shared Data Block` `block` and a nonnegative integer `i`. It performs the following steps:

1. Assert: `block` is a `Shared Data Block`.
2. Assert: `i` and `i+3` are valid byte offsets within the memory of `block`.
3. Assert: `i` is divisible by 4.
4. Return the `WaiterList` that is referenced by the pair `(block, i)`.

### 24.4.1.4 EnterCriticalSection( `WL` )

The abstract operation EnterCriticalSection takes one argument, a `WaiterList` `WL`. It performs the following steps:

1. Assert: The calling agent is not in the critical section for any `WaiterList`.
2. Wait until no agent is in the critical section for `WL`, then enter the critical section for `WL` (without allowing any other agent to enter).

### 24.4.1.5 LeaveCriticalSection( `WL` )
The abstract operation LeaveCriticalSection takes one argument, a WaiterList \( WL \). It performs the following steps:

1. Assert: The calling agent is in the critical section for \( WL \).
2. Leave the critical section for \( WL \).

### 24.4.1.6 AddWaiter( \( WL, W \) )

The abstract operation AddWaiter takes two arguments, a WaiterList \( WL \) and an agent signifier \( W \). It performs the following steps:

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 \( WL \).

### 24.4.1.7 RemoveWaiter( \( WL, W \) )

The abstract operation RemoveWaiter takes two arguments, a WaiterList \( WL \) and an agent signifier \( W \). It performs the following steps:

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 \( WL \).

### 24.4.1.8 RemoveWaiters( \( WL, c \) )

The abstract operation RemoveWaiters takes two arguments, a WaiterList \( WL \) and nonnegative integer \( c \). It performs the following steps:

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. Subtract 1 from \( c \).
5. Return \( L \).

### 24.4.1.9 Suspend( \( WL, W, timeout \) )
The abstract operation Suspend takes three arguments, a `WaiterList WL`, an `agent` signifier `W`, and a nonnegative, non-`NaN` Number `timeout`. It performs the following steps:

1. Assert: The calling `agent` is in the critical section for `WL`.
2. Assert: `W` is equal to `AgentSignifier()`.
3. Assert: `W` is on the list of waiters in `WL`.
4. Assert: `AgentCanSuspend()` is `true`.
5. Perform `LeaveCriticalSection(WL)` and suspend `W` for up to `timeout` milliseconds, performing the combined operation in such a way that a wakeup that arrives after the critical section is exited but before the suspension takes effect is not lost. `W` can wake up either because the timeout expired or because it was woken explicitly by another `agent` calling `WakeWaiter(WL, W)`, and not for any other reasons at all.
6. Perform `EnterCriticalSection(WL)`.
7. If `W` was woken explicitly by another `agent` calling `WakeWaiter(WL, W)`, return `true`.
8. Return `false`.

### 24.4.1.10 `WakeWaiter(WL, W)`

The abstract operation `WakeWaiter` takes two arguments, a `WaiterList WL` and an `agent` signifier `W`. It performs the following steps:

1. Assert: The calling `agent` is in the critical section for `WL`.
2. Assert: `W` is on the list of waiters in `WL`.
3. Wake the `agent W`.

**NOTE** The embedding may delay waking `W`, e.g. for resource management reasons, but `W` must eventually be woken in order to guarantee forward progress.

### 24.4.1.11 `AtomicReadModifyWrite(typedArray, index, value, op)`

The abstract operation `AtomicReadModifyWrite` takes four arguments, `typedArray`, `index`, `value`, and a pure combining operation `op`. The pure combining operation `op` takes two `List` of byte values arguments and returns a `List` of byte values. The 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:

1. Let `buffer` be `ValidateSharedIntegerTypedArray(typedArray)`. 
2. Let `i` be `ValidateAtomicAccess(typedArray, index)`. 
3. Let `v` be `ToInteger(value)`.
4. Let `arrayTypeName` be `typedArray. [[Typed_ARRAY_NAME]]`. 
5. Let `elementSize` be the Number value of the Element Size value specified in Table 50 for `arrayTypeName`. 
6. Let `elementType` be the String value of the Element Type value in Table 50 for `arrayTypeName`. 

---

1. **NOTE** The embedding may delay waking `W`, e.g. for resource management reasons, but `W` must eventually be woken in order to guarantee forward progress.

---

**Table 50**

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Element Type</th>
<th>Element Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Array</td>
<td>Int8</td>
<td>1</td>
</tr>
<tr>
<td>Uint8Array</td>
<td>Uint8</td>
<td>1</td>
</tr>
<tr>
<td>Int16Array</td>
<td>Int16</td>
<td>2</td>
</tr>
<tr>
<td>Uint16Array</td>
<td>Uint16</td>
<td>2</td>
</tr>
<tr>
<td>Int32Array</td>
<td>Int32</td>
<td>4</td>
</tr>
<tr>
<td>Uint32Array</td>
<td>Uint32</td>
<td>4</td>
</tr>
<tr>
<td>Float32Array</td>
<td>Float32</td>
<td>4</td>
</tr>
<tr>
<td>Float64Array</td>
<td>Float64</td>
<td>8</td>
</tr>
</tbody>
</table>

---

**Table 50**

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Element Type</th>
<th>Element Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Array</td>
<td>Int8</td>
<td>1</td>
</tr>
<tr>
<td>Uint8Array</td>
<td>Uint8</td>
<td>1</td>
</tr>
<tr>
<td>Int16Array</td>
<td>Int16</td>
<td>2</td>
</tr>
<tr>
<td>Uint16Array</td>
<td>Uint16</td>
<td>2</td>
</tr>
<tr>
<td>Int32Array</td>
<td>Int32</td>
<td>4</td>
</tr>
<tr>
<td>Uint32Array</td>
<td>Uint32</td>
<td>4</td>
</tr>
<tr>
<td>Float32Array</td>
<td>Float32</td>
<td>4</td>
</tr>
<tr>
<td>Float64Array</td>
<td>Float64</td>
<td>8</td>
</tr>
</tbody>
</table>

---

1. **NOTE** The embedding may delay waking `W`, e.g. for resource management reasons, but `W` must eventually be woken in order to guarantee forward progress.
7. Let offset be typedArray.[[ByteOffset]].
8. Let indexedPosition be \((i \times elementSize) + offset\).
9. Return `GetModifySetValueInBuffer(buffer, indexedPosition, elementType, v, op)`.

### 24.4.1.12 AtomicLoad( `typedArray`, `index` )

The abstract operation AtomicLoad takes two arguments, `typedArray`, `index`. The operation atomically loads a value and returns the loaded value. It performs the following steps:

1. Let `buffer` be ? `ValidateSharedIntegerTypedArray(typedArray)`.
2. Let `i` be ? `ValidateAtomicAccess(typedArray, index)`.
3. Let `arrayTypeName` be `typedArray`.[[TypedArrayName]].
4. Let `elementSize` be the Number value of the Element Size value specified in Table 50 for `arrayTypeName`.
5. Let `elementType` be the String value of the Element Type value in Table 50 for `arrayTypeName`.
6. Let `offset` be `typedArray`.[[ByteOffset]].
7. Let `indexedPosition` be \((i \times elementSize) + offset\).
8. Return `GetValueFromBuffer(buffer, indexedPosition, elementType, true, "SeqCst")`.

### 24.4.2 Atomics.add( `typedArray`, `index`, `value` )

Let `add` denote a semantic function of two `List` of byte values arguments that applies the addition operation to the Number values corresponding to the `List` of byte values arguments and returns a `List` of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return ? `AtomicReadModifyWrite(typedArray, index, value, add)`.

### 24.4.3 Atomics.and( `typedArray`, `index`, `value` )

Let `and` denote a semantic function of two `List` of byte values arguments that applies the bitwise-and operation element-wise to the two arguments and returns a `List` of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return ? `AtomicReadModifyWrite(typedArray, index, value, and)`.

### 24.4.4 Atomics.compareExchange( `typedArray`, `index`, `expectedValue`, `replacementValue` )

The following steps are taken:

1. Let \textit{buffer} be \? \textit{ValidateSharedIntegerTypedArray(\textit{typedArray})}.
2. Let \textit{i} be \? \textit{ValidateAtomicAccess(\textit{typedArray, index})}.
3. Let \textit{expected} be \? \textit{ToInteger(\textit{expectedValue})}.
4. Let \textit{replacement} be \? \textit{ToInteger(\textit{replacementValue})}.
5. Let \textit{arrayTypeName} be \textit{typedArray.}[\textit{[TypedArrayName]}].
6. Let \textit{elementType} be the String value of the Element Type value in \textit{Table 50} for \textit{arrayTypeName}.
7. Let \textit{isLittleEndian} be the value of the \textit{[[LittleEndian]]} field of the surrounding agent’s Agent Record.
8. Let \textit{expectedBytes} be \textit{NumberToRawBytes(\textit{elementType, expected, isLittleEndian})}.
9. Let \textit{elementSize} be the Number value of the Element Size value specified in \textit{Table 50} for \textit{arrayTypeName}.
10. Let \textit{offset} be \textit{typedArray.}[\textit{[ByteOffset]}].
11. Let \textit{indexedPosition} be \((i \times \textit{elementSize}) + \textit{offset})
12. Let \textit{compareExchange} denote a semantic function of two \textit{List} of byte values arguments that returns the second argument if the first argument is element-wise equal to \textit{expectedBytes}.
13. Return \textit{GetModifySetValueInBuffer(buffer, indexedPosition, elementType, replacement, compareExchange)}.

\textbf{24.4.5} \textbf{Atomics.exchange( \textit{typedArray, index, value} )}

Let \textit{second} denote a semantic function of two \textit{List} of byte values arguments that returns its second argument.

The following steps are taken:

1. Return \? \textit{AtomicReadModifyWrite(\textit{typedArray, index, value, second})}.

\textbf{24.4.6} \textbf{Atomics.isLockFree( \textit{size} )}

The following steps are taken:

1. Let \textit{n} be \? \textit{ToInteger(\textit{size})}.
2. Let \textit{AR} be the Agent Record of the surrounding agent.
3. If \textit{n} equals 1, return \textit{AR.}[\textit{[IsLockFree1]}].
4. If \textit{n} equals 2, return \textit{AR.}[\textit{[IsLockFree2]}].
5. If \textit{n} equals 4, return \textit{true}. 
6. Return \textit{false}.

\textbf{NOTE} \textit{Atomics.isLockFree()} is an optimization primitive. The intuition is that if the atomic step of an atomic primitive (\textit{compareExchange, load, store, add, sub, and, or, xor, or exchange}) on a datum of size \textit{n} bytes will be performed without the calling agent.
acquiring a lock outside the \( n \) bytes comprising the datum, then \( \text{Atomics.isLockFree}(n) \) will return \text{true}. High-performance algorithms will use \( \text{Atomics.isLockFree} \) to determine whether to use locks or atomic operations in critical sections. If an atomic primitive is not lock-free then it is often more efficient for an algorithm to provide its own locking.

\( \text{Atomics.isLockFree}(4) \) always returns \text{true} as that can be supported on all known relevant hardware. Being able to assume this will generally simplify programs.

### 24.4.7 \text{Atomics.load( } \text{typedArray, index } \text{ )}

The following steps are taken:

1. Return \( ? \text{AtomicLoad(typedArray, index)} \).

### 24.4.8 \text{Atomics.or( } \text{typedArray, index, value } \text{ )}

Let \text{or} denote a semantic function of two \text{List} of byte values arguments that applies the bitwise-or operation element-wise to the two arguments and returns a \text{List} of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return \( ? \text{AtomicReadModifyWrite(typedArray, index, value, or)} \).

### 24.4.9 \text{Atomics.store( } \text{typedArray, index, value } \text{ )}

The following steps are taken:

1. Let \( \text{buffer} \) be \( \text{ValidateSharedIntegerTypedArray(typedArray)} \).
2. Let \( i \) be \( \text{ValidateAtomicAccess(typedArray, index)} \).
3. Let \( v \) be \( \text{ToInteger(value)} \).
4. Let \( \text{arrayTypeName} \) be \( \text{typedArray.}[[\text{TypedArrayName}]] \).
5. Let \( \text{elementSize} \) be the Number value of the Element Size value specified in \text{Table 50} for \( \text{arrayTypeName} \).
6. Let \( \text{elementType} \) be the String value of the Element Type value in \text{Table 50} for \( \text{arrayTypeName} \).
7. Let \( \text{offset} \) be \( \text{typedArray.}[[\text{ByteOffset}]] \).
8. Let \( \text{indexedPosition} \) be \( (i \times \text{elementSize}) + \text{offset} \).
9. Perform \( \text{SetValueInBuffer(buffer, indexedPosition, elementType, v, true, "SeqCst"}) \).
10. Return \( v \).

### 24.4.10 \text{Atomics.sub( } \text{typedArray, index, value } \text{ )}
Let `subtract` denote a semantic function of two `List` of byte values arguments that applies the subtraction operation to the Number values corresponding to the `List` of byte values arguments and returns a `List` of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return `AtomicReadModifyWrite(typedArray, index, value, subtract)`.

### 24.4.11 Atomics.wait( `typedArray, index, value, timeout` )

`Atomics.wait` puts the calling agent in a wait queue and puts it to sleep until it is awoken or the sleep times out. The following steps are taken:

1. Let `buffer` be ? `ValidateSharedIntegerTypedArray(typedArray, true)`.  
2. Let `i` be ? `ValidateAtomicAccess(typedArray, index)`.  
3. Let `v` be ? `ToInt32(value)`.  
4. Let `q` be ? `ToNumber(timeout)`.  
5. If `q` is `NaN`, let `t` be `+∞`, else let `t` be `max(q, 0)`.  
6. Let `B` be `AgentCanSuspend()`.  
7. If `B` is `false`, throw a `TypeError` exception.  
8. Let `block` be `buffer.[[ArrayBufferData]]`.  
9. Let `offset` be `typedArray.[[ByteOffset]]`.  
10. Let `indexedPosition` be `(i * 4) + offset`.  
11. Let `WL` be `GetWaiterList(block, indexedPosition)`.  
12. Perform `EnterCriticalSection(WL)`.  
13. Let `w` be `! AtomicLoad(typedArray, i)`.  
14. If `v` is not equal to `w`, then  
   a. Perform `LeaveCriticalSection(WL)`.  
   b. Return the String "not-equal".  
15. Let `W` be `AgentSignifier()`.  
17. Let `awoken` be `Suspend(WL, W, t)`.  
19. Perform `LeaveCriticalSection(WL)`.  
20. If `awoken` is `true`, return the String "ok".  
21. Return the String "timed-out".

### 24.4.12 Atomics.wake( `typedArray, index, count` )

`Atomics.wake` wakes up some agents that are sleeping in the wait queue. The following steps are taken:
1. Let `buffer` be \(?\) ValidateSharedIntegerTypedArray(`typedArray`, true).
2. Let `i` be \(?\) ValidateAtomicAccess(`typedArray`, `index`).
3. If `count` is `undefined`, let `c` be +\(\infty\).
4. Else,
   a. Let `intCount` be `ToIntInteger(count)`.
   b. Let `c` be \(\max(intCount, 0)\).
5. Let `block` be `buffer`.[[ArrayBufferData]].
6. Let `offset` be `typedArray`.[[ByteOffset]].
7. Let `indexedPosition` be \((i \times 4) + offset\).
8. Let `WL` be `GetWaiterList(block, indexedPosition)`.
9. Let `n` be 0.
10. Perform `EnterCriticalSection(WL)`.
11. Let `S` be `RemoveWaiters(WL, c)`.
12. Repeat, while `S` is not an empty `List`,
   a. Let `W` be the first `agent` in `S`.
   b. Remove `W` from the front of `S`.
   c. Perform `WakeWaiter(WL, W)`.
   d. Add 1 to `n`.
13. Perform `LeaveCriticalSection(WL)`.

### 24.4.13 Atomics.xor( `typedArray`, `index`, `value` )

Let `xor` denote a semantic function of two `List` of byte values arguments that applies the bitwise-xor operation element-wise to the two arguments and returns a `List` of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return \(?\) AtomicReadModifyWrite(`typedArray`, `index`, `value`, `xor`).

### 24.4.14 Atomics [ @@toStringTag ]

The initial value of the `@@toStringTag` property is the String value \"Atomics\".

This property has the attributes { {[Writable]}: `false`, {[Enumerable]}: `false`, {[Configurable]}: `true` }.

### 24.5 The JSON Object
The JSON object is the `%JSON% intrinsic object and the initial value of the JSON property of the global object. The JSON object is a single ordinary object that contains two functions, parse and stringify, that are used to parse and construct JSON texts. The JSON Data Interchange Format is defined in ECMA-404. The JSON interchange format used in this specification is exactly that described by ECMA-404.

Conforming implementations of JSON.parse and JSON.stringify must support the exact interchange format described in the ECMA-404 specification without any deletions or extensions to the format.

The value of the [[Prototype]] internal slot of the JSON object is the intrinsic object %ObjectPrototype%. The value of the [[Extensible]] internal slot of the JSON object is set to true.

The JSON object does not have a [[Construct]] internal method; it is not possible to use the JSON object as a constructor with the new operator.

The JSON object does not have a [[Call]] internal method; it is not possible to invoke the JSON object as a function.

### 24.5.1 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 JText be ? ToString(text).
2. Parse JText interpreted as UTF-16 encoded Unicode points (6.1.4) as a JSON text as specified in ECMA-404. Throw a SyntaxError exception if JText is not a valid JSON text as defined in that specification.
3. Let scriptText be the result of concatenating "(" , JText , and ");".
4. Let completion be the result of parsing and evaluating scriptText as if it was the source text of an ECMAScript Script, but using the alternative definition of DoubleStringCharacter provided below. The extended PropertyDefinitionEvaluation semantics defined in B.3.1 must not be used during the evaluation.
5. Let unfiltered be completion.[[Value]].
6. Assert: unfiltered is either a String, Number, Boolean, Null, or an Object that is defined by either an ArrayLiteral or an Object Literal.
7. If $\text{Callable}(\text{reviver})$ is $\text{true}$, then
   a. Let $\text{root}$ be $\text{ObjectCreate}(%\text{ObjectPrototype}%)$.
   b. Let $\text{rootName}$ be the empty String.
   c. Let $\text{status}$ be $\text{CreateDataProperty}(\text{root}, \text{rootName}, \text{unfiltered})$.
   d. Assert: $\text{status}$ is $\text{true}$.
   e. Return $\text{InternalizeJSONProperty}(\text{root}, \text{rootName})$.
8. Else,
   a. Return $\text{unfiltered}$.

The $\text{length}$ property of the $\text{parse}$ function is 2.

JSON allows Unicode code units 0x2028 (LINE SEPARATOR) and 0x2029 (PARAGRAPH SEPARATOR) to directly appear in String literals without using an escape sequence. This is enabled by using the following alternative definition of $\text{DoubleStringCharacter}$ when parsing $\text{scriptText}$ in step 4:

$$\text{DoubleStringCharacter} ::
\text{SourceCharacter} \text{ but not one of } " \text{ or } \text\backslash \text{ or U+0000 through U+001F}
\text\backslash \text\text{EscapeSequence}$$

- The SV of $\text{DoubleStringCharacter} :: \text{SourceCharacter}$ but not one of "$" or \ or U+0000 through U+001F is the $\text{UTF16Encoding}$ of the code point value of $\text{SourceCharacter}$.

**NOTE** Valid JSON text is a subset of the ECMAScript $\text{PrimaryExpression}$ syntax as modified by Step 4 above. Step 2 verifies that $\text{JText}$ conforms to that subset, and step 6 verifies that that parsing and evaluation returns a value of an appropriate type.

### 24.5.1.1 Runtime Semantics: $\text{InternalizeJSONProperty}(\text{holder}, \text{name})$

The abstract operation $\text{InternalizeJSONProperty}$ is a recursive abstract operation that takes two parameters: a $\text{holder}$ object and the String $\text{name}$ of a property in that object. $\text{InternalizeJSONProperty}$ uses the value of $\text{reviver}$ that was originally passed to the above parse function.

1. Let $\text{val}$ be $\text{Get}(\text{holder}, \text{name})$.
2. If $\text{Type}(\text{val})$ is Object, then
   a. Let $\text{isArray}$ be $\text{IsArray}(\text{val})$.
   b. If $\text{isArray}$ is $\text{true}$, then
      i. Let $i$ be 0.
      ii. Let $\text{len}$ be $\text{ToLength}(\text{Get}(\text{val}, "\text{length}"))$.
      iii. Repeat, while $i < \text{len}$,
         1. Let $\text{newElement}$ be $\text{InternalizeJSONProperty}(\text{val}, !\text{ToString}(i))$.
         2. If $\text{newElement}$ is $\text{undefined}$, then
            a. Perform $\text{val}.[[\text{Delete}]](!\text{ToString}(i))$. 

3. Else,
a. Perform ? CreateDataProperty(val, !ToString(I), newElement).
b. NOTE: This algorithm intentionally does not throw an exception if CreateDataProperty returns false.

c. Else,
i. Let keys be ? EnumerableOwnProperties(val, "key").
ii. For each String P in keys, do
  1. Let newElement be ? InternalizeJSONProperty(val, P).
  2. If newElement is undefined, then
  3. Else,
     b. NOTE: This algorithm intentionally does not throw an exception if CreateDataProperty returns false.

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 In the case where there are duplicate name Strings within an object, lexically preceding values for the same key shall be overwritten.

24.5.2 JSON.stringify ( value [, replacer [, space ] ] )

The stringify function returns a String in UTF-16 encoded JSON format representing an ECMAScript value. It can take three parameters. The value parameter is an ECMAScript value, which is usually an object or array, although it can also be a String, Boolean, Number or null. The optional replacer parameter is either a function that alters the way objects and arrays are stringified, or an array of Strings and Numbers that acts as 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. Let ReplacerFunction be replacer.
b. Else,
   i. Let isArray be ? isArray(replacer).
   ii. If isArray is true, then
      1. Let PropertyList be a new empty List.
      2. Let len be ? ToLength(? Get(replacer, "length")).
      3. Let k be 0.
      4. Repeat, while k<len,
         a. Let v be ? Get(replacer, ! ToString(k)).
         b. Let item be undefined.
         c. If type(v) is String, let item be v.
         d. Else if type(v) is Number, let item be ! ToString(v).
         e. Else if type(v) is Object, then
            i. If v has a [[StringData]] or [[NumberData]] internal slot, let item be ? ToString(v).
         f. If item is not undefined and item is not currently an element of PropertyList, then
            i. Append item to the end of PropertyList.
      g. Let k be k+1.
5. If type(space) is Object, then
   a. If space has a [[NumberData]] internal slot, then
      i. Let space be ? ToNumber(space).
   b. Else if space has a [[StringData]] internal slot, then
      i. Let space be ? ToString(space).
6. If type(space) is Number, then
   a. Let space be min(10, ToInteger(space)).
   b. Set gap to a string containing space occurrences of code unit 0x0020 (SPACE). This will be the
      empty string if space is less than 1.
7. Else if type(space) is String, then
   a. If the number of elements in space is 10 or less, set gap to space; otherwise set gap to a string
      consisting of the first 10 elements of space.
8. Else,
   a. Set gap to the empty string.
9. Let wrapper be ObjectCreate(%ObjectPrototype%).
10. Let status be CreateDataProperty(wrapper, the empty String, value).
11. Assert status is true.
12. Return ? SerializeJSONProperty(the empty String, wrapper).

The length property of the stringify function is 3.

NOTE 1 JSON structures are allowed to be nested to any depth, but they must be acyclic. If value is
or contains a cyclic structure, then the `stringify` function must throw a `TypeError` exception. This is an example of a value that cannot be stringified:

```javascript
a = [];  
a[0] = a;  
my_text = JSON.stringify(a); // This must throw a TypeError.
```

**NOTE 2** Symbolic primitive values are rendered as follows:

- The `null` value is rendered in JSON text as the String `null`.
- The `undefined` value is not rendered.
- The `true` value is rendered in JSON text as the String `true`.
- The `false` value is rendered in JSON text as the String `false`.

**NOTE 3** String values are wrapped in QUOTATION MARK (" code units. The code units " and \ are escaped with \ prefixes. Control characters code units are replaced with escape sequences `uHHHH`, or with the shorter forms, \b (BACKSPACE), \f (FORM FEED), \n (LINE FEED), \r (CARRIAGE RETURN), \t (CHARACTER TABULATION).

**NOTE 4** Finite numbers are stringified as if by calling `ToString(number)`. NaN and Infinity regardless of sign are represented as the String `null`.

**NOTE 5** Values that do not have a JSON representation (such as `undefined` and functions) do not produce a String. Instead they produce the `undefined` value. In arrays these values are represented as the String `null`. In objects an unrepresentable value causes the property to be excluded from stringification.

**NOTE 6** An object is rendered as U+007B (LEFT CURLY BRACKET) followed by zero or more properties, separated with a U+002C (COMMA), closed with a U+007D (RIGHT CURLY BRACKET). A property is a quoted String representing the key or property name, a U+003A (COLON), and then the stringified property value. An array is rendered as an opening U+005B (LEFT SQUARE BRACKET followed by zero or more values, separated with a U+002C (COMMA), closed with a U+005D (RIGHT SQUARE BRACKET).

### 24.5.2.1 Runtime Semantics: SerializeJSONProperty (key, holder)

The abstract operation SerializeJSONProperty with arguments `key`, and `holder` has access to `ReplacerFunction` from the invocation of the `stringify` method. Its algorithm is as follows:

1. Let `value` be ? `Get(holder, key)`.
2. If `Type(value)` is Object, then
   a. Let `toJSON` be ? `Get(value, "toJSON")`.
b. If `IsCallable(toJSON)` is `true`, then
   i. Set `value` to `Call(toJSON, value, «key»)`.

3. If `ReplacerFunction` is not `undefined`, then
   a. Set `value` to `Call(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]]`.

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 Object and `IsCallable(value)` is `false`, then
    a. Let `isArray` be ? `IsArray(value)`.
    b. If `isArray` is `true`, return ? `SerializeJSONArray(value)`.
    c. Return ? `SerializeJSONObject(value)`.

11. Return `undefined`.

### 24.5.2.2 Runtime Semantics: QuoteJSONString (value)

The abstract operation `QuoteJSONString` with argument `value` wraps a String value in QUOTATION MARK code units and escapes certain other code units within it.

1. Let `product` be code unit 0x0022 (QUOTATION MARK).
2. For each code unit `C` in `value`, do
   a. If `C` is 0x0022 (QUOTATION MARK) or 0x005C (REVERSE SOLIDUS), then
      i. Set `product` to the concatenation of `product` and code unit 0x005C (REVERSE SOLIDUS).
      ii. Set `product` to the concatenation of `product` and `C`.
   b. Else if `C` is 0x0008 (BACKSPACE), 0x000C (FORM FEED), 0x000A (LINE FEED), 0x000D (CARRIAGE RETURN), or 0x0009 (CHARACTER TABULATION), then
      i. Set `product` to the concatenation of `product` and code unit 0x005C (REVERSE SOLIDUS).
      ii. Let `abbrev` be the String value corresponding to the value of `C` as follows:

      | Character       | Abbreviation |
      |-----------------|--------------|
      | BACKSPACE       | "b"          |
      | FORM FEED (FF)  | "f"          |
iii. Set `product` to the concatenation of `product` and `abbrev`.

c. Else if `C` has a code unit value less than 0x0020 (SPACE), then
   i. Set `product` to the concatenation of `product` and code unit 0x005C (REVERSE SOLIDUS).
   ii. Set `product` to the concatenation of `product` and "u".
   iii. Let `hex` be the string result of converting the numeric code unit value of `C` to a String of four lowercase hexadecimal digits.
   iv. Set `product` to the concatenation of `product` and `hex`.

d. Else,
   i. Set `product` to the concatenation of `product` and `C`.

3. Set `product` to the concatenation of `product` and code unit 0x0022 (QUOTATION MARK).
4. Return `product`.

24.5.2.3 Runtime Semantics: `SerializeJSONObject` (value)

The abstract operation `SerializeJSONObject` with argument `value` serializes an object. It has access to the `stack`, `indent`, `gap`, and `PropertyList` values of the current invocation of the `stringify` method.

1. If `stack` contains `value`, throw a `TypeError` exception because the structure is cyclical.
2. Append `value` to `stack`.
3. Let `stepback` be `indent`.
4. Set `indent` to the concatenation of `indent` and `gap`.
5. If `PropertyList` is not `undefined`, then
   a. Let `K` be `PropertyList`.
6. Else,
   a. Let `K` be `EnumerableOwnProperties(value, "key")`.
7. Let `partial` be a new empty List.
8. For each element `P` of `K`, do
   a. Let `strP` be `SerializeJSONProperty(P, value)`.
   b. If `strP` is not `undefined`, then
      i. Let `member` be `QuoteJSONObject(P)`.
      ii. Set `member` to the concatenation of `member` and the String ": ".
      iii. If `gap` is not the empty String, then
          1. Set `member` to the concatenation of `member` and code unit 0x0020 (SPACE).
          iv. Set `member` to the concatenation of `member` and `strP`.
      v. Append `member` to `partial`.
9. If `partial` is empty, then
   a. Let `final` be "{"".
10. Else,
   a. If gap is the empty String, then
      i. Let properties be a String formed by concatenating all the element Strings of partial with each adjacent pair of Strings separated with code unit 0x002C (COMMA). A comma is not inserted either before the first String or after the last String.
      ii. Let final be the result of concatenating "{", properties, and "}".
   b. Else gap is not the empty String,
      i. Let separator be the result of concatenating code unit 0x002C (COMMA), code unit 0x000A (LINE FEED), and indent.
      ii. Let properties be a String formed by concatenating all the element Strings of partial with each adjacent pair of Strings separated with separator. The separator String is not inserted either before the first String or after the last String.
      iii. Let final be the result of concatenating "{", code unit 0x000A (LINE FEED), indent, properties, code unit 0x000A (LINE FEED), stepback, and "}".
   11. Remove the last element of stack.
   12. Set indent to stepback.

24.5.2.4 Runtime Semantics: SerializeJSONArray (value)

The abstract operation SerializeJSONArray with argument value serializes an array. It has access to the stack, indent, and gap values of the current invocation of the stringify method.

1. If stack contains value, throw a TypeError exception because the structure is cyclical.
2. Append value to stack.
3. Let stepback be indent.
4. Set indent to the concatenation of indent and gap.
5. Let partial be a new empty List.
6. Let len be ? ToLength(? Get(value, "length")).
7. Let index be 0.
8. Repeat, while index < len
   a. Let strP be ? SerializeJSONProperty(! ToString(index), value).
   b. If strP is undefined, then
      i. Append "null" to partial.
   c. Else,
      i. Append strP to partial.
      d. Increment index by 1.
9. If partial is empty, then
   a. Let final be "[]".
10. Else,
   a. If gap is the empty String, then
i. Let properties be a String formed by concatenating all the element Strings of partial with each adjacent pair of Strings separated with code unit 0x002C (COMMA). A comma is not inserted either before the first String or after the last String.
ii. Let final be the result of concatenating "[", properties, and "]".

b. Else,
i. Let separator be the result of concatenating code unit 0x002C (COMMA), code unit 0x000A (LINE FEED), and indent.
ii. Let properties be a String formed by concatenating all the element Strings of partial with each adjacent pair of Strings separated with separator. The separator String is not inserted either before the first String or after the last String.
iii. Let final be the result of concatenating "[", code unit 0x000A (LINE FEED), indent, properties, code unit 0x000A (LINE FEED), stepback, and "]".

11. Remove the last element of stack.
12. Set indent to stepback.

NOTE The representation of arrays includes only the elements between zero and array.length - 1 inclusive. Properties whose keys are not array indexes are excluded from the stringification. An array is stringified as an opening LEFT SQUARE BRACKET, elements separated by COMMA, and a closing RIGHT SQUARE BRACKET.

24.5.3 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 }.

25 Control Abstraction Objects

25.1 Iteration

25.1.1 Common Iteration Interfaces

An interface is a set of property keys whose associated values match a specific specification. Any object that provides all the properties as described by an interface’s specification conforms to that interface. An interface is not represented by a distinct object. There may be many separately implemented objects that conform to any interface. An individual object may conform to multiple interfaces.
25.1.1.1 The *Iterable* Interface

The *Iterable* interface includes the property described in Table 53:

<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 <em>Iterator</em> object.</td>
<td>The returned object must conform to the <em>Iterator</em> interface.</td>
</tr>
</tbody>
</table>

25.1.1.2 The *Iterator* Interface

An object that implements the *Iterator* interface must include the property in Table 54. Such objects may also implement the properties in Table 55.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>A function that returns an <em>IteratorResult</em> object.</td>
<td>The returned object must conform to the <em>IteratorResult</em> interface. If a previous call to the <em>next</em> method of an <em>Iterator</em> has returned an <em>IteratorResult</em> object whose <em>done</em> property is <em>true</em>, then all subsequent calls to the <em>next</em> method of that object should also return an <em>IteratorResult</em> object whose <em>done</em> property is <em>true</em>. 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>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>A function that returns an <em>IteratorResult</em> object.</td>
<td>The returned object must conform to the <em>IteratorResult</em> interface. Invoking this method notifies the <em>Iterator</em> object that the caller does not intend to make any more <em>next</em> method calls to the <em>Iterator</em>. The returned <em>IteratorResult</em> object will typically have a <em>done</em> property whose value is <em>true</em>, and a <em>value</em> property with the value passed as the argument of the <em>return</em> method. However, this requirement is not enforced.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>A function that returns an <em>IteratorResult</em> object.</td>
<td>The returned object must conform to the <em>IteratorResult</em> interface. Invoking this method notifies the <em>Iterator</em> object that the caller has detected an error condition. The argument may be used to identify the error condition and</td>
</tr>
</tbody>
</table>
an IteratorResult object. The argument may be used to identify the error condition and typically will be an exception object. A typical response is to throw the value passed as the argument. If the method does not throw, the returned IteratorResult object will typically have a done property whose value is true.

NOTE 2 Typically callers of these methods should check for their existence before invoking them. Certain ECMAScript language features including for-of, yield*, and array destructuring call these methods after performing an existence check. Most ECMAScript library functions that accept Iterable objects as arguments also conditionally call them.

25.1.1.3 The IteratorResult Interface

The IteratorResult interface includes the properties listed in Table 56:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>done</td>
<td>Either true or false.</td>
<td>This is the result status of an iterator next method call. If the end of the iterator was reached done is true. If the end was not reached done is false and a value is available. If a done property (either own or inherited) does not exist, it is consider to have the value false.</td>
</tr>
<tr>
<td>value</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, value is undefined. In that case, the value property may be absent from the conforming object if it does not inherit an explicit value property.</td>
</tr>
</tbody>
</table>

25.1.2 The %IteratorPrototype% Object

The value of the [[Prototype]] internal slot of the %IteratorPrototype% object is the intrinsic object %ObjectPrototype%. The %IteratorPrototype% object is an ordinary object. The initial value of the [[Extensible]] internal slot of the %IteratorPrototype% object is true.

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]()))
```

### 25.1.2.1 `%IteratorPrototype%` [ @@iterator ] ()

The following steps are taken:

1. Return the `this` value.

The value of the `name` property of this function is "`[Symbol.iterator]`".

### 25.2 GeneratorFunction Objects

GeneratorFunction objects are functions that are usually created by evaluating `GeneratorDeclarations`, `GeneratorExpressions`, and `GeneratorMethods`. They may also be created by calling the `%GeneratorFunction%` intrinsic.

---

**Figure 2 (Informative): Generator Objects Relationships**

![Diagram showing the relationships between various types of objects relevant to generator functions and objects. The diagram includes nodes for `Object.prototype`, `Function.prototype`, `%Generator%`, `%GeneratorPrototype%`, and `%IteratorPrototype%`, along with arrows indicating the relationships and properties such as `constructor`, `hasOwnProperty`, and `toStringTag`.](image-url)
25.2.1 The GeneratorFunction Constructor

The `GeneratorFunction` constructor is the `%GeneratorFunction%` intrinsic. When `GeneratorFunction` is called as a function rather than as a constructor, it creates and initializes a new `GeneratorFunction` object. Thus the function call `GeneratorFunction (...)` is equivalent to the object creation expression `new GeneratorFunction (...)` with the same arguments.

`GeneratorFunction` 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.

25.2.1.1 `GeneratorFunction ( p1, p2, ..., pn, body )`

The last argument specifies the body (executable code) of a generator function; any preceding arguments specify formal parameters.

When the `GeneratorFunction` function is called with some arguments `p1, p2, ..., pn, body` (where `n` might be 0, that is, there are no “p” arguments, and where `body` might also not be provided), the following steps are taken:

1. Let `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 19.2.1.1.

25.2.2 Properties of the GeneratorFunction Constructor

The `GeneratorFunction` constructor is a standard built-in function object that inherits from the `Function` constructor. The value of the `[[Prototype]]` internal slot of the `GeneratorFunction` constructor
is the intrinsic object %Function%.

The value of the [[Extensible]] internal slot of the GeneratorFunction constructor is true.

The value of the name property of the GeneratorFunction is "GeneratorFunction".

The GeneratorFunction constructor has the following properties:

### 25.2.2.1 GeneratorFunction.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

### 25.2.2.2 GeneratorFunction.prototype

The initial value of GeneratorFunction.prototype is the intrinsic object %Generator%. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 25.2.3 Properties of the GeneratorFunction Prototype Object

The GeneratorFunction prototype object is an ordinary object. It is not a function object and does not have an [[ECMAScriptCode]] internal slot or any other of the internal slots listed in Table 27 or Table 57. In addition to being the value of the prototype property of the %GeneratorFunction% intrinsic, it is the %Generator% intrinsic (see Figure 2).

The value of the [[Prototype]] internal slot of the GeneratorFunction prototype object is the %FunctionPrototype% intrinsic object. The initial value of the [[Extensible]] internal slot of the GeneratorFunction prototype object is true.

#### 25.2.3.1 GeneratorFunction.prototype.constructor

The initial value of GeneratorFunction.prototype.constructor is the intrinsic object %GeneratorFunction%.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

#### 25.2.3.2 GeneratorFunction.prototype.prototype

The value of GeneratorFunction.prototype.prototype is the %GeneratorPrototype% intrinsic object.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }. 
25.2.3.3 GeneratorFunction.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "GeneratorFunction".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

25.2.4 GeneratorFunction Instances

Every GeneratorFunction instance is an ECMAScript function object and has the internal slots listed in Table 27. The value of the [[FunctionKind]] internal slot for all such instances is "generator".

Each GeneratorFunction instance has the following own properties:

25.2.4.1 length

The specification for the length property of Function instances given in 19.2.4.1 also applies to GeneratorFunction instances.

25.2.4.2 name

The specification for the name property of Function instances given in 19.2.4.2 also applies to GeneratorFunction instances.

25.2.4.3 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.

25.3 Generator Objects

A Generator object is an instance of a generator function and conforms to both the Iterator and Iterable interfaces.
Generator instances directly inherit properties from the object that is the value of the `prototype` property of the Generator function that created the instance. Generator instances indirectly inherit properties from the Generator Prototype intrinsic, `%GeneratorPrototype%.

### 25.3.1 Properties of Generator Prototype

The Generator prototype object is the `%GeneratorPrototype%` intrinsic. It is also the initial value of the `prototype` property of the `%Generator%` intrinsic (the GeneratorFunction.prototype).

The Generator prototype is an ordinary object. It is not a Generator instance and does not have a `[[GeneratorState]]` internal slot.

The value of the `[[Prototype]]` internal slot of the Generator prototype object is the intrinsic object `%IteratorPrototype%`. The initial value of the `[[Extensible]]` internal slot of the Generator prototype object is `true`.

All Generator instances indirectly inherit properties of the Generator prototype object.

#### 25.3.1.1 Generator.prototype.constructor

The initial value of `Generator.prototype.constructor` is the intrinsic object `%Generator%.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

#### 25.3.1.2 Generator.prototype.next ( value )

The `next` method performs the following steps:

1. Let `g` be the `this` value.
2. Return `? GeneratorResume(g, value).

#### 25.3.1.3 Generator.prototype.return ( value )

The `return` method performs the following steps:

1. Let `g` be the `this` value.
2. Let `C` be `Completion([[Type]]: return, [[Value]]: `value`, [[Target]]: empty).`.
3. Return `? GeneratorResumeAbrupt(g, C).

#### 25.3.1.4 Generator.prototype.throw ( exception )

The `throw` method performs the following steps:
1. Let $g$ be the **this** value.
2. Let $C$ be Completion({[Type]: throw, [Value]: exception, [Target]: empty}).
3. Return ? GeneratorResumeAbrupt($g$, $C$).

### 25.3.1.5 Generator.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the String value "Generator".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

### 25.3.2 Properties of Generator Instances

Generator instances are initially created with the internal slots described in Table 57.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[GeneratorState]]</td>
<td>The current execution state of the generator. The possible values are: undefined, &quot;suspendedStart&quot;, &quot;suspendedYield&quot;, &quot;executing&quot;, and &quot;completed&quot;.</td>
</tr>
<tr>
<td>[[GeneratorContext]]</td>
<td>The execution context that is used when executing the code of this generator.</td>
</tr>
</tbody>
</table>

### 25.3.3 Generator Abstract Operations

#### 25.3.3.1 GeneratorStart (generator, generatorBody)

The abstract operation GeneratorStart with arguments *generator* and *generatorBody* performs the following steps:

1. Assert: The value of *generator*.[[GeneratorState]] is undefined.
2. Let *genContext* be the running execution context.
3. Set the Generator component of *genContext* to *generator*.
4. Set the code evaluation state of *genContext* such that when evaluation is resumed for that execution context the following steps will be performed:
   a. Let result be the result of evaluating *generatorBody*.
   b. Assert: If we return here, the generator either threw an exception or performed either an implicit or explicit return.
   c. Remove *genContext* from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.
   d. Set *generator*.[[GeneratorState]] to "completed".
Once a generator enters the "completed" state it never leaves it and its associated execution context is never resumed. Any execution state associated with generator can be discarded at this point.

If `result.([Type])` is normal, let `resultValue` be undefined.

Else if `result.([Type])` is return, let `resultValue` be `result.[[Value]].`

h. Else,
   i. Assert: `result.([Type])` is throw.
   ii. Return `Completion(result)`.
   iii. Return `CreateIterResultObject(resultValue, true)`.

5. Set `generator.[[GeneratorContext]]` to `genContext`.

6. Set `generator.[[GeneratorState]]` to "suspendedStart".

7. Return `NormalCompletion(undefined)`.

### 25.3.3.2 GeneratorValidate (generator)

The abstract operation `GeneratorValidate` with argument `generator` performs the following steps:

1. If `Type(generator)` is not Object, throw a `TypeError` exception.
2. If `generator` does not have a `[[GeneratorState]]` internal slot, throw a `TypeError` exception.
3. Assert: `generator` also has a `[[GeneratorContext]]` internal slot.
4. Let `state` be `generator.[[GeneratorState]]`.
5. If `state` is "executing", throw a `TypeError` exception.
6. Return `state`.

### 25.3.3.3 GeneratorResume (generator, value)

The abstract operation `GeneratorResume` with arguments `generator` and `value` performs the following steps:

1. Let `state` be `? GeneratorValidate(generator)`.
2. If `state` is "completed", return `CreateIterResultObject(undefined, true)`.
3. Assert: `state` is either "suspendedStart" or "suspendedYield".
4. Let `genContext` be `generator.[[GeneratorContext]]`.
5. Let `methodContext` be the running execution context.
7. Set `generator.[[GeneratorState]]` to "executing".
8. Push `genContext` onto the execution context stack; `genContext` is now the running execution context.
9. Resume the suspended evaluation of `genContext` using `NormalCompletion(value)` as the result of the operation that suspended it. Let `result` be the value returned by the resumed computation.
10. Assert: When we return here, `genContext` has already been removed from the execution context stack and `methodContext` is the currently running execution context.
11. Return \text{Completion}(result).

25.3.3.4 \textbf{GeneratorResumeAbrupt (generator, abruptCompletion)}

The abstract operation GeneratorResumeAbrupt with arguments \textit{generator} and \textit{abruptCompletion} performs the following steps:

1. Let \textit{state} be \textit{GeneratorValidate(generator)}.
2. If \textit{state} is "suspendedStart", then
   a. Set \textit{generator}.[[GeneratorState]] to "completed".
   b. Once a generator enters the "completed" state it never leaves it and its associated execution context is never resumed. Any execution state associated with \textit{generator} can be discarded at this point.
   c. Set \textit{state} to "completed".
3. If \textit{state} is "completed", then
   a. If \textit{abruptCompletion}.[[Type]] is return, then
      i. Return \text{CreateIterResultObject}(\textit{abruptCompletion}.[[Value]], false).
   b. Return \text{Completion}(\textit{abruptCompletion}).
4. Assert: \textit{state} is "suspendedYield".
5. Let \textit{genContext} be \textit{generator}.[[GeneratorContext]].
6. Let \textit{methodContext} be the running execution context.
7. \text{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. 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 \text{Completion}(\textit{result}).

25.3.3.5 \textbf{GeneratorYield (iterNextObj)}

The abstract operation GeneratorYield with argument \textit{iterNextObj} performs the following steps:

1. Assert: \textit{iterNextObj} is an Object that implements the \textit{IteratorResult} interface.
2. Let \textit{genContext} be the running execution context.
3. Assert: \textit{genContext} is the execution context of a generator.
4. Let \textit{generator} be the value of the Generator component of \textit{genContext}.
5. Set \textit{generator}.[[GeneratorState]] to "suspendedYield".
6. Remove `genContext` from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.

7. Set the code evaluation state of `genContext` such that when evaluation is resumed with a `Completion resumptionValue` the following steps will be performed:
   a. Return `resumptionValue`.
   b. NOTE: This returns to the evaluation of the `YieldExpression` that originally called this abstract operation.

8. Return `NormalCompletion(iterNextObj)`.

9. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of `genContext`.

25.4 Promise Objects

A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.

Any Promise object is in one of three mutually exclusive states: *fulfilled*, *rejected*, and *pending*:

- A promise `p` is fulfilled if `p.then(f, r)` will immediately enqueue a Job to call the function `f`.
- A promise `p` is rejected if `p.then(f, r)` will immediately enqueue a Job to call the function `r`.
- A promise is pending if it is neither fulfilled nor rejected.

A promise is said to be *settled* if it is not pending, i.e. if it is either fulfilled or rejected.

A promise is *resolved* if it is settled or if it has been “locked in” to match the state of another promise. Attempting to resolve or reject a resolved promise has no effect. A promise is *unresolved* if it is not resolved. An unresolved promise is always in the pending state. A resolved promise may be pending, fulfilled or rejected.

25.4.1 Promise Abstract Operations

25.4.1.1 PromiseCapability Records

A PromiseCapability is a `Record` value used to encapsulate a promise object along with the functions that are capable of resolving or rejecting that promise object. PromiseCapability Records are produced by the `NewPromiseCapability` abstract operation.

PromiseCapability Records have the fields listed in Table 58.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>

Table 58: PromiseCapability `Record` Fields
If `IfAbruptRejectPromise (value, capability)` is a short hand for a sequence of algorithm steps that use a PromiseCapability Record. An algorithm step of the form:

1. `IfAbruptRejectPromise(value, capability)`.

means the same thing as:

1. If `value` is an abrupt completion, then
   a. Perform `Call(capability.([Reject]), undefined, «value,[Value]»)`.
   b. Return `capability.([Promise])`.
2. Else if `value` is a Completion Record, let `value` be `value.[Value]`.

### 25.4.1.1 PromiseReaction Records

The PromiseReaction is a `Record` value used to store information about how a promise should react when it becomes resolved or rejected with a given value. PromiseReaction records are created by the `PerformPromiseThen` abstract operation, and are used by a PromiseReactionJob.

PromiseReaction records have the fields listed in Table 59.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Capability]]</code></td>
<td>A PromiseCapability Record</td>
<td>The capabilities of the promise for which this record provides a reaction handler.</td>
</tr>
<tr>
<td><code>[[Type]]</code></td>
<td>Either &quot;Fulfill&quot; or &quot;Reject&quot;</td>
<td>The <code>[[Type]]</code> is used when <code>[[Handler]]</code> is <code>undefined</code> to allow for behaviour specific to the settlement type.</td>
</tr>
<tr>
<td><code>[[Handler]]</code></td>
<td>A function object or <code>undefined</code></td>
<td>The function that should be applied to the incoming value, and whose return value will govern what happens to the derived promise. If <code>[[Handler]]</code> is <code>undefined</code>, a function that depends on the value of <code>[[Type]]</code> will be used instead.</td>
</tr>
</tbody>
</table>
### 25.4.1.3 CreateResolvingFunctions (promise)

When CreateResolvingFunctions is performed with argument promise, the following steps are taken:

1. Let `alreadyResolved` be a new Record {[[Value]]: false}.
2. Let `resolve` be a new built-in function object as defined in Promise Resolve Functions (25.4.1.3.2).
3. Set `resolve`.[[Promise]] to promise.
4. Set `resolve`.[[AlreadyResolved]] to `alreadyResolved`.
5. Let `reject` be a new built-in function object as defined in Promise Reject Functions (25.4.1.3.1).
6. Set `reject`.[[Promise]] to promise.
7. Set `reject`.[[AlreadyResolved]] to `alreadyResolved`.
8. Return a new Record {[[Resolve]]: `resolve`, [[Reject]]: `reject`}.

#### 25.4.1.3.1 Promise Reject Functions

A promise reject function is an anonymous built-in function that has `[[Promise]]` and `[[AlreadyResolved]]` internal slots.

When a promise reject function `F` is called with argument `reason`, the following steps are taken:

1. Assert: `F` has a `[[Promise]]` internal slot whose value is an Object.
2. Let `promise` be `F`.[[Promise]].
3. Let `alreadyResolved` be `F`.[[AlreadyResolved]].
4. If `alreadyResolved`.[[Value]] is true, return undefined.
5. Set `alreadyResolved`.[[Value]] to true.
6. Return `RejectPromise(promise, reason)`.

The `length` property of a promise reject function is 1.

#### 25.4.1.3.2 Promise Resolve Functions

A promise resolve function is an anonymous built-in function that has `[[Promise]]` and `[[AlreadyResolved]]` internal slots.

When a promise resolve function `F` is called with argument `resolution`, the following steps are taken:

1. Assert: `F` has a `[[Promise]]` internal slot whose value is an Object.
2. Let `promise` be `F`.[[Promise]].
3. Let `alreadyResolved` be `F`.[[AlreadyResolved]].
4. If `alreadyResolved`.[[Value]] is true, return undefined.
5. Set `alreadyResolved`.[[Value]] to true.
6. If `SameValue(resolution, promise)` is true, then
   a. Let `selfResolutionError` be a newly created `TypeError` object.
   b. Return `RejectPromise(promise, selfResolutionError)`.
7. If \(\text{Type}(\text{resolution})\) is not \text{Object}, then
   a. Return \(\text{FulfillPromise}(\text{promise}, \text{resolution})\).
8. Let \(\text{then}\) be \(\text{Get}(\text{resolution}, \text{"then"})\).
9. If \(\text{then}\) is an abrupt completion, then
   a. Return \(\text{RejectPromise}(\text{promise}, \text{then}.[[\text{Value}]])\).
10. Let \(\text{thenAction}\) be \(\text{then}.[[\text{Value}]]\).
11. If \(\text{IsCallable}(\text{thenAction})\) is \text{false}, then
    a. Return \(\text{FulfillPromise}(\text{promise}, \text{resolution})\).
12. Perform \(\text{EnqueueJob}(\text{"PromiseJobs"}, \text{PromiseResolveThenableJob}, \text{« promise, resolution, thenAction »})\).
13. Return \text{undefined}.

The \text{length} property of a promise resolve function is 1.

### 25.4.1.4 FulfillPromise (\(\text{promise, value}\))

When the FulfillPromise abstract operation is called with arguments \(\text{promise}\) and \(\text{value}\), the following steps are taken:

1. Assert: The value of \(\text{promise}.[[\text{PromiseState}]]\) is "\text{pending}".
2. Let \(\text{reactions}\) be \(\text{promise}.[[\text{PromiseFulfillReactions}]]\).
3. Set \(\text{promise}.[[\text{PromiseResult}]]\) to \(\text{value}\).
4. Set \(\text{promise}.[[\text{PromiseFulfillReactions}]]\) to \text{undefined}.
5. Set \(\text{promise}.[[\text{PromiseRejectReactions}]]\) to \text{undefined}.
6. Set \(\text{promise}.[[\text{PromiseState}]]\) to "\text{fulfilled}".
7. Return \(\text{TriggerPromiseReactions}(\text{reactions, value})\).

### 25.4.1.5 NewPromiseCapability (\(C\))

The abstract operation \text{NewPromiseCapability} takes a constructor function, and attempts to use that constructor function in the fashion of the built-in \text{Promise} constructor to create a Promise object and extract its resolve and reject functions. The promise plus the resolve and reject functions are used to initialize a new PromiseCapability \text{Record} which is returned as the value of this abstract operation.

1. If \(\text{IsConstructor}(C)\) is \text{false}, throw a \text{TypeError} exception.
2. NOTE: \(C\) is assumed to be a constructor function that supports the parameter conventions of the \text{Promise} constructor (see 25.4.3.1).
3. Let \(\text{promiseCapability}\) be a new PromiseCapability \{ [[Promise]]: \text{undefined}, [[Resolve]]: \text{undefined}, [[Reject]]: \text{undefined} \}.
4. Let \(\text{executor}\) be a new built-in function object as defined in \text{GetCapabilitiesExecutor Functions}.
5. Set \(\text{executor}.[[\text{Capability}]]\) to \(\text{promiseCapability}\).
6. Let \(\text{promise}\) be \? \text{Construct}(C, \text{« executor »}).
7. If `IsCallable(promiseCapability.([Resolve]])` is `false`, throw a `TypeError` exception.
8. If `IsCallable(promiseCapability.([Reject]])` is `false`, throw a `TypeError` exception.
9. Set `promiseCapability.([Promise])` to `promise`.
10. 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.

### 25.4.1.5.1 GetCapabilitiesExecutor Functions

A GetCapabilitiesExecutor function is an anonymous built-in function that has a `[[Capability]]` internal slot. When a GetCapabilitiesExecutor function `F` is called with arguments `resolve` and `reject`, the following steps are taken:

1. Assert: `F` has a `[[Capability]]` internal slot whose value is a PromiseCapability `Record`.
2. Let `promiseCapability` be `F.([Capability])`.
3. If `promiseCapability.([Resolve])` is not `undefined`, throw a `TypeError` exception.
4. If `promiseCapability.([Reject])` is not `undefined`, throw a `TypeError` exception.
5. Set `promiseCapability.([Resolve])` to `resolve`.
6. Set `promiseCapability.([Reject])` to `reject`.
7. Return `undefined`.

The `length` property of a GetCapabilitiesExecutor function is 2.

### 25.4.1.6 IsPromise (x)

The abstract operation IsPromise checks for the promise brand on an object.

1. If `Type(x)` is not `Object`, return `false`.
2. If `x` does not have a `[[PromiseState]]` internal slot, return `false`.
3. Return `true`.

### 25.4.1.7 RejectPromise (promise, reason)

When the RejectPromise abstract operation is called with arguments `promise` and `reason`, the following steps are taken:

1. Assert: The value of `promise.([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)`.

### 25.4.1.8 TriggerPromiseReactions (`reactions, argument`) 

The abstract operation `TriggerPromiseReactions` takes a collection of `PromiseReactionRecords` and enqueues a new job for each record. Each such job processes the `[[Type]]` and `[[Handler]]` of the `PromiseReactionRecord`, and if the `[[Handler]]` is a function, calls it passing the given argument. If the `[[Handler]]` is `undefined`, the behaviour is determined by the `[[Type]]`.

1. For each `reaction` in `reactions`, in original insertion order, do
2. Return `undefined`.

### 25.4.1.9 HostPromiseRejectionTracker (`promise, operation`) 

`HostPromiseRejectionTracker` is an implementation-defined abstract operation that allows host environments to track promise rejections.

An implementation of `HostPromiseRejectionTracker` must complete normally in all cases. The default implementation of `HostPromiseRejectionTracker` is to unconditionally return an empty normal completion.

---

**NOTE 1**

`HostPromiseRejectionTracker` is called in two scenarios:

- When a promise is rejected without any handlers, it is called with its `operation` argument set to "reject".
- When a handler is added to a rejected promise for the first time, it is called with its `operation` argument set to "handle".

A typical implementation of `HostPromiseRejectionTracker` might try to notify developers of unhandled rejections, while also being careful to notify them if such previous notifications are later invalidated by new handlers being attached.

---

**NOTE 2**

If `operation` is "handle", an implementation should not hold a reference to `promise` in a way that would interfere with garbage collection. An implementation may hold a reference to `promise` if `operation` is "reject", since it is expected that rejections will be rare and not on hot code paths.

### 25.4.2 Promise Jobs
25.4.2.1 PromiseReactionJob (reaction, argument)

The job PromiseReactionJob with parameters reaction and argument applies the appropriate handler to the incoming value, and uses the handler’s return value to resolve or reject the derived promise associated with that handler.

1. Assert: reaction is a PromiseReaction Record.
2. Let promiseCapability be reaction.[[Capability]].
3. Let type be reaction.[[Type]].
4. Let handler be reaction.[[Handler]].
5. If handler is undefined, then
   a. If type is "Fulfill", let handlerResult be NormalCompletion(argument).
   b. Else,
      i. Assert: type is "Reject".
      ii. Let handlerResult be Completion{[[Type]]: throw, [[Value]]: argument, [[Target]]: empty}.
6. Else, let handlerResult be Call(handler, undefined, « argument »).
7. If handlerResult is an abrupt completion, then
   a. Let status be Call(promiseCapability.[[Reject]], undefined, « handlerResult.[[Value]] »).
8. Else,
   a. Let status be Call(promiseCapability.[[Resolve]], undefined, « handlerResult.[[Value]] »).
9. Return Completion(status).

25.4.2.2 PromiseResolveThenableJob (promiseToResolve, thenable, then)

The job PromiseResolveThenableJob with parameters promiseToResolve, thenable, and then performs the following steps:

1. Let resolvingFunctions be CreateResolvingFunctions(promiseToResolve).
2. Let thenCallResult be Call(then, thenable, « resolvingFunctions.[[Resolve]], resolvingFunctions.[[Reject]] »).
3. If thenCallResult is an abrupt completion, then
   a. Let status be Call(resolvingFunctions.[[Reject]], undefined, « thenCallResult.[[Value]] »).
   b. Return Completion(status).
4. Return Completion(thenCallResult).

NOTE This job uses the supplied thenable and its then method to resolve the given promise. This process must take place as a Job to ensure that the evaluation of the then method occurs after evaluation of any surrounding code has completed.

25.4.3 The Promise Constructor
The Promise constructor is the `%Promise%` intrinsic object and the initial value of the `Promise` property of the global object. When called as a constructor it creates and initializes a new Promise object. `Promise` is not intended to be called as a function and will throw an exception when called in that manner.

The `Promise` constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `Promise` behaviour must include a `super` call to 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.

### 25.4.3.1 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, "%PromisePrototype%", «
   [[PromiseState]], [[PromiseResult]], [[PromiseFulfillReactions]], [[PromiseRejectReactions]],
   [[PromiselsHandled]] »).
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`.[[PromiselsHandled]] 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.

### 25.4.4 Properties of the Promise Constructor

The value of the `[[Prototype]]` internal slot of the `Promise` constructor is the intrinsic object `%FunctionPrototype%`.

The Promise constructor has the following properties:

#### 25.4.4.1 Promise.all (iterable)

The `all` function returns a new promise which is fulfilled with an array of fulfillment values for the passed promises, or rejects with the reason of the first passed promise that rejects. It resolves all elements of the passed iterable to promises as it runs this algorithm.

1. Let `C` be the `this` value.
2. If `Type(C)` is not `Object`, throw a `TypeError` exception.
3. Let `promiseCapability` be `? NewPromiseCapability(C)`.
4. Let `iterator` be `GetIterator(iterable)`.
5. If `AbruptRejectPromise(iterator, promiseCapability)`.
6. Let `iteratorRecord` be `Record` `{ [[Iterator]]: iterator, [[Done]]: false }`.
7. Let `result` be `PerformPromiseAll(iteratorRecord, C, promiseCapability)`.
8. If `result` is an abrupt completion, then
   a. If `iteratorRecord.[[Done]]` is `false`, let `result` be `IteratorClose(iterator, result)`.
   b. If `AbruptRejectPromise(result, promiseCapability)`.
9. Return `Completion(result)`.

---

**NOTE**

The `all` function requires its `this` value to be a constructor function that supports the parameter conventions of the `Promise` constructor.

#### 25.4.4.1.1 Runtime Semantics: PerformPromiseAll(`iteratorRecord`, `constructor`, `resultCapability`)
1. Assert: constructor is a constructor function.
2. Assert: resultCapability is a PromiseCapability Record.
3. Let values be a new empty List.
4. Let remainingElementsCount be a new Record { [[Value]]: 1 }.
5. Let index be 0.
6. Repeat,
   a. Let next be IteratorStep(iteratorRecord.([[Iterator]])).
   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 ? Invoke(constructor, "resolve", « nextValue »).
      j. Let resolveElement be a new built-in function object as defined in Promise.all Resolve Element Functions.
      k. Set resolveElement.([[AlreadyCalled]]) to a new Record { [[Value]]: false }.
      l. Set resolveElement.([[Index]]) to index.
      m. Set resolveElement.([[Values]]) to values.
      n. Set resolveElement.([[Capability]]) to resultCapability.
      o. Set resolveElement.([[RemainingElements]]) to remainingElementsCount.
      p. Set remainingElementsCount.([[Value]]) to remainingElementsCount.([[Value]]) + 1.
      q. Perform ? Invoke(nextPromise, "then", « resolveElement, resultCapability.([[Resolve]]) »).
      r. Set index to index + 1.

25.4.4.1.2 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 F is called with argument x, the following steps are taken:

   1. Let alreadyCalled be F.([[AlreadyCalled]]).
2. If `alreadyCalled.[[Value]]` is `true`, return `undefined`.
3. Set `alreadyCalled.[[Value]]` to `true`.
4. Let `index` be `F.[[Index]]`.
5. Let `values` be `F.[[Values]]`.
7. Let `remainingElementsCount` be `F.[[RemainingElements]]`.
8. Set `values[index]` to `x`.
9. Set `remainingElementsCount.[[Value]]` to `remainingElementsCount.[[Value]] - 1`.
10. If `remainingElementsCount.[[Value]]` is 0, then
    a. Let `valuesArray` be `CreateArrayFromList(values)`.
    b. Return `? Call(promiseCapability.[[Resolve]], undefined, « valuesArray »)`.
11. Return `undefined`.

The `length` property of a `Promise.all` resolve element function is 1.

### 25.4.4.2 Promise.prototype

The initial value of `Promise.prototype` is the intrinsic object `%PromisePrototype%`.

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 25.4.4.3 Promise.race (iterable)

The `race` function returns a new promise which is settled in the same way as the first passed promise to settle. It resolves all elements of the passed `iterable` to promises as it runs this algorithm.

1. Let `C` be the `this` value.
2. If `Type(C)` is not Object, throw a `TypeError` exception.
3. Let `promiseCapability` be `? NewPromiseCapability(C)`.
4. Let `iterator` be `GetIterator(iterator)`.
5. If `AbruptRejectPromise(iterator, promiseCapability)`.
6. Let `iteratorRecord` be `Record { [[Iterator]]: iterator, [[Done]]: false }`.
7. Let `result` be `PerformPromiseRace(iteratorRecord, C, promiseCapability)`.
8. If `result` is an abrupt completion, then
    a. If `iteratorRecord.[[Done]]` is `false`, let `result` be `IteratorClose(iterator, 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
25.4.3.1 Runtime Semantics: PerformPromiseRace (iteratorRecord, constructor, resultCapability)

When the PerformPromiseRace abstract operation is called with arguments iteratorRecord, constructor, and resultCapability, the following steps are taken:

1. Assert: constructor is a constructor function.
2. Assert: resultCapability is a PromiseCapability Record.
3. Repeat,
   a. Let next be IteratorStep(iteratorRecord.[[Iterator]]).
   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 ? Invoke(constructor, "resolve", « nextValue »).
      i. Perform ? Invoke(nextPromise, "then", « resultCapability.[[Resolve]], resultCapability.[[Reject]] »).

25.4.4 Promise.reject (r)

The reject function returns a new promise rejected with the passed argument.

1. Let C be the this value.
2. If Type(C) is not Object, throw a TypeError exception.
4. Perform ? Call(promiseCapability.[[Reject]], undefined, « r »).
5. 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.

25.4.5 Promise.resolve (x)

The resolve function returns either a new promise resolved with the passed argument, or the argument itself if the argument is a promise produced by this constructor.
1. Let \( C \) be the \textbf{this} value.
2. If \texttt{Type\((C)\)} is not \texttt{Object}, throw a \textbf{TypeError} exception.
3. If \texttt{IsPromise\((x)\)} is \texttt{true}, then
   a. Let \texttt{xConstructor} be \texttt{? Get\((x, \"constructor\")\)}.
   b. If \texttt{SameValue\((xConstructor, C)\)} is \texttt{true}, return \( x \).
4. Let \texttt{promiseCapability} be \texttt{? NewPromiseCapability\((C)\)}.
5. Perform \texttt{? Call\((promiseCapability.\[[\text{Resolve}], \text{undefined}, \text{"x\"}\])\).}
6. Return \texttt{promiseCapability.\[[\text{Promise}]\].}

\textbf{NOTE} The \texttt{resolve} function expects its \texttt{this} value to be a constructor function that supports the parameter conventions of the \texttt{Promise} constructor.

\section*{25.4.4.6 \texttt{get Promise} [ @@species ]}

\texttt{Promise[@@species]} is an accessor property whose set accessor function is \texttt{undefined}. Its get accessor function performs the following steps:

1. Return the \texttt{this} value.

The value of the \texttt{name} property of this function is \texttt{"get [Symbol.species]"}.

\textbf{NOTE} Promise prototype methods normally use their \texttt{this} object’s constructor to create a derived object. However, a subclass constructor may over-ride that default behaviour by redefining its \texttt{@@species} property.

\section*{25.4.5 Properties of the Promise Prototype Object}

The Promise prototype object is the intrinsic object \%\texttt{PromisePrototype}\%. The value of the \texttt{[[Prototype]]} internal slot of the Promise prototype object is the intrinsic object \%\texttt{ObjectPrototype}\%. The Promise prototype object is an ordinary object. It does not have a \texttt{[[PromiseState]]} internal slot or any of the other internal slots of Promise instances.

\subsection*{25.4.5.1 Promise.prototype.catch ( \texttt{onRejected} )}

When the \texttt{catch} method is called with argument \texttt{onRejected}, the following steps are taken:

1. Let \texttt{promise} be the \texttt{this} value.
2. Return \texttt{? Invoke\((promise, \"then\", \texttt{undefined, onRejected\")} \).}

\subsection*{25.4.5.2 Promise.prototype.constructor}

The initial value of \texttt{Promise.prototype.constructor} is the intrinsic object \%\texttt{Promise}\%. 
25.4.5.3 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).

25.4.5.3.1 PerformPromiseThen (promise, onFulfilled, onRejected, resultCapability)

The abstract operation PerformPromiseThen performs the “then” operation on promise using onFulfilled and onRejected as its settlement actions. The result is resultCapability’s promise.

1. Assert: IsPromise(promise) is true.
2. Assert: resultCapability is a PromiseCapability Record.
3. If IsCallable(onFulfilled) is false, then
   a. Set onFulfilled to undefined.
4. If IsCallable(onRejected) is false, then
   a. Set onRejected to undefined.
5. Let fulfillReaction be the PromiseReaction { [[Capability]]: resultCapability, [[Type]]: "Fulfill", [[Handler]]: onFulfilled }.
6. Let rejectReaction be the PromiseReaction { [[Capability]]: resultCapability, [[Type]]: "Reject", [[Handler]]: onRejected }.
7. 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]].
8. Else if promise.[[PromiseState]] is "fulfilled", then
   a. Let value be promise.[[PromiseResult]].
   b. Perform EnqueueJob("PromiseJobs", PromiseReactionJob, « fulfillReaction, value »).
9. Else,
   a. Assert: The value of promise.[[PromiseState]] is "rejected".
   b. Let reason be promise.[[PromiseResult]].
   c. If promise.[[PromisesAreHandled]] is false, perform HostPromiseRejectionTracker(promise, "handle").
   d. Perform EnqueueJob("PromiseJobs", PromiseReactionJob, « rejectReaction, reason »).
10. Set promise.[[PromisesAreHandled]] to true.
11. Return resultCapability.[[Promise]].

25.4.5.4 Promise.prototype [ @@toStringTag ]
The initial value of the `@@toStringTag` property is the String value "Promise".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

### 25.4.6 Properties of Promise Instances

Promise instances are ordinary objects that inherit properties from the Promise prototype object (the intrinsic, `%PromisePrototype%`). Promise instances are initially created with the internal slots described in Table 60.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[PromiseState]]</code></td>
<td>A String value that governs how a promise will react to incoming calls to its <code>then</code> method. The possible values are: &quot;pending&quot;, &quot;fulfilled&quot;, and &quot;rejected&quot;.</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 &quot;pending&quot;.</td>
</tr>
<tr>
<td><code>[[PromiseFulfillReactions]]</code></td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the &quot;pending&quot; state to the &quot;fulfilled&quot; state.</td>
</tr>
<tr>
<td><code>[[PromiseRejectReactions]]</code></td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the &quot;pending&quot; state to the &quot;rejected&quot; state.</td>
</tr>
<tr>
<td><code>[[PromisesIsHandled]]</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>

### 25.5 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.

#### 25.5.1 The AsyncFunction Constructor

The `AsyncFunction` constructor is the `%AsyncFunction%` intrinsic object and is a subclass of `Function`. When `AsyncFunction` is called as a function rather than as a constructor, it creates and initializes a new AsyncFunction object. Thus the function call `AsyncFunction(...)` is equivalent to the object creation expression `new AsyncFunction(...)` with the same arguments.
The AsyncFunction 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 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.

25.5.1 AsyncFunction\((p1, p2, \ldots, pn, body)\)

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 \(p1, p2, \ldots, pn, body\) (where \(n\) might be 0, that is, there are no \(p\) arguments, and where \(body\) might also not be provided), the following steps are taken:

1. Let \(C\) be the active function object.
2. Let \(args\) be the argumentsList that was passed to this function by [[Call]] or [[Construct]].
3. Return CreateDynamicFunction\((C, NewTarget, "async", args)\).

**NOTE** See NOTE for 19.2.1.1.

25.5.2 Properties of the AsyncFunction Constructor

The AsyncFunction constructor is a standard built-in function object that inherits from the Function constructor. The value of the [[Prototype]] internal slot of the AsyncFunction constructor is the intrinsic object %Function%.

The value of the [[Extensible]] internal slot of the AsyncFunction constructor is true.

The value of the name property of the AsyncFunction is "AsyncFunction".

The AsyncFunction constructor has the following properties:

25.5.2.1 AsyncFunction.length

This is a data property with a value of 1. This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true \}.

25.5.2.2 AsyncFunction.prototype

The initial value of AsyncFunction.prototype is the intrinsic object %AsyncFunctionPrototype%. This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \}. 
The AsyncFunction prototype object is an ordinary object. It is not a function object and does not have an [[ECMAScriptCode]] internal slot or any other of the internal slots listed in Table 27. In addition to being the value of the prototype property of the %AsyncFunction% intrinsic, it is the %AsyncFunctionPrototype% intrinsic.

The value of the [[Prototype]] internal slot of the AsyncFunction prototype object is the %FunctionPrototype% intrinsic object. The initial value of the [[Extensible]] internal slot of the AsyncFunction prototype object is true.

25.5.3.1 AsyncFunction.prototype.constructor

The initial value of `AsyncFunction.prototype.constructor` is the intrinsic object `%AsyncFunction%`. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

25.5.3.2 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 }.

25.5.4 AsyncFunction Instances

Every AsyncFunction instance is an ECMAScript function object and has the internal slots listed in Table 27. The value of the [[FunctionKind]] internal slot for all such instances is "async". 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 constructable.

Each AsyncFunction instance has the following own properties:

25.5.4.1 length

The specification for the length property of Function instances given in 19.2.4.1 also applies to AsyncFunction instances.

25.5.4.2 name

The specification for the name property of Function instances given in 19.2.4.2 also applies to AsyncFunction instances.
25.5.5 Async Functions Abstract Operations

25.5.5.1 AsyncFunctionCreate ( \textit{kind}, \textit{parameters}, \textit{body}, \textit{Scope}, \textit{Strict} )

The abstract operation AsyncFunctionCreate requires the arguments: \textit{kind} which is one of (Normal, Method, Arrow), a parameter list Parse Node specified by \textit{parameters}, a body Parse Node specified by \textit{body}, a Lexical Environment specified by \textit{Scope}, and a Boolean flag \textit{Strict}. AsyncFunctionCreate performs the following steps:

1. Let \textit{functionPrototype} be the intrinsic object \%AsyncFunctionPrototype\%.
2. Let \textit{F} be \textbf{! FunctionAllocate}(\textit{functionPrototype}, \textit{Strict}, "async").

25.5.5.2 AsyncFunctionStart ( \textit{promiseCapability}, \textit{asyncFunctionBody} )

1. Let \textit{runningContext} be the running execution context.
2. Let \textit{asyncContext} be a copy of \textit{runningContext}.
3. Set the code evaluation state of \textit{asyncContext} such that when evaluation is resumed for that \textit{execution context} the following steps will be performed:
   a. Let \textit{result} be the result of evaluating \textit{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 \textit{asyncContext} from the \textit{execution context stack} and restore the \textit{execution context} that is at the top of the \textit{execution context stack} as the \textit{running execution context}.
   d. If \textit{result}.[[Type]] is normal, then
      i. Perform \textbf{! Call}(\textit{promiseCapability}.[[Resolve]], \textit{undefined}, «undefined»).
   e. Else if \textit{result}.[[Type]] is return, then
      i. Perform \textbf{! Call}(\textit{promiseCapability}.[[Resolve]], \textit{undefined}, «result}.[[Value]]»).
   f. Else,
      i. Assert: \textit{result}.[[Type]] is throw.
      ii. Perform \textbf{! Call}(\textit{promiseCapability}.[[Reject]], \textit{undefined}, «result}.[[Value]]»).
   g. Return.
4. Push \textit{asyncContext} onto the \textit{execution context stack}; \textit{asyncContext} is now the \textit{running execution context}.
5. Resume the suspended evaluation of \textit{asyncContext}. Let \textit{result} be the value returned by the resumed computation.
6. Assert: When we return here, \textit{asyncContext} has already been removed from the \textit{execution context stack} and \textit{runningContext} is the currently \textit{running execution context}.
7. Assert: \textit{result} is a normal completion with a value of \textit{undefined}. The possible sources of completion values are AsyncFunctionAwait or, if the async function doesn’t await anything, the step 3.g above.
8. Return.
25.5.5.3 AsyncFunctionAwait (value)

1. Let asyncContext be the running execution context.
2. Let promiseCapability be ! NewPromiseCapability(%Promise%).
3. Let resolveResult be ! Call(promiseCapability.([[Resolve]], undefined, «value»)).
4. Let onFulfilled be a new built-in function object as defined in AsyncFunction Awaited Fulfilled.
5. Let onRejected be a new built-in function object as defined in AsyncFunction Awaited Rejected.
7. Set onRejected.([[AsyncContext]]) to asyncContext.
8. Let throwawayCapability be ! NewPromiseCapability(%Promise%).
9. Set throwawayCapability.([[Promise]].[[PromisesShandled]]) to true.
10. Perform ! PerformPromiseThen(promiseCapability.([[Promise]], onFulfilled, onRejected, throwawayCapability).
11. 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.
12. Set the code evaluation state of asyncContext such that when evaluation is resumed with a Completion resumptionValue the following steps will be performed:
   a. Return resumptionValue.
13. Return.

NOTE The return value of this abstract operation is unused. The interesting return is that of resumptionValue being returned to the AwaitExpression that originally called this abstract operation.

25.5.5.4 AsyncFunction Awaited Fulfilled

An AsyncFunction Awaited Fulfilled function is an anonymous built-in function that has an [[AsyncContext]] internal slot. The value of the [[AsyncContext]] internal slot is the execution context that will be restored when the function is called.

When an AsyncFunction Awaited Fulfilled function F is called with argument value, the following steps are taken:

1. Let asyncContext be F.([[AsyncContext]])
2. Let prevContext be the running execution context.
4. Push asyncContext onto the execution context stack; asyncContext is now the running execution context.
5. Resume the suspended evaluation of asyncContext using NormalCompletion(value) as the result of the operation that suspended it. Let result be the value returned by the resumed computation.
6. Assert: When we reach this step, asyncContext has already been removed from the execution context stack and prevContext is the currently running execution context.
Return `Completion(result)`.

### 25.5.5.5 AsyncFunction Awaited Rejected

An AsyncFunction Awaited Rejected function is an anonymous built-in function that has an `[[AsyncContext]]` internal slot. The value of the `[[AsyncContext]]` internal slot is the `execution context` that will be restored when the function is called.

When an AsyncFunction Awaited Rejected function `F` is called with argument `reason`, the following steps are taken:

1. Let `asyncContext` be `F.[[AsyncContext]]`.
2. Let `prevContext` be the running execution context.
4. Push `asyncContext` onto the execution context stack; `asyncContext` is now the running execution context.
5. Resume the suspended evaluation of `asyncContext` using `Completion{[[Type]]: throw, [[Value]]: reason, [[Target]]: empty}` as the result of the operation that suspended it. Let `result` be the value returned by the resumed computation.
6. Assert: When we reach this step, `asyncContext` has already been removed from the execution context stack and `prevContext` is the currently running execution context.
7. Return `Completion(result)`.

### 26 Reflection

#### 26.1 The Reflect Object

The Reflect object is the `%Reflect%` intrinsic object and the initial value of the `Reflect` property of the global object. The Reflect object is an ordinary object.

The value of the `[[Prototype]]` internal slot of the Reflect object is the intrinsic object `%ObjectPrototype%`.

The Reflect object is not a function object. It does not have a `[[Construct]]` internal method; it is not possible to use the Reflect object as a constructor with the `new` operator. The Reflect object also does not have a `[[Call]]` internal method; it is not possible to invoke the Reflect object as a function.

#### 26.1.1 Reflect.apply (`target, thisArgument, argumentsList`)
When the **apply** function is called with arguments *target*, *thisArgument*, and *argumentsList*, the following steps are taken:

1. If `IsCallable(target)` is **false**, throw a **TypeError** exception.
2. Let `args` be `? CreateListFromArrayLike(argumentsList)`.
3. Perform `PrepareForTailCall()`.
4. Return `? Call(target, thisArgument, args)`.

### 26.1.2 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, let `newTarget` be `target`.
3. Else if `IsConstructor(newTarget)` is **false**, throw a **TypeError** exception.
4. Let `args` be `? CreateListFromArrayLike(argumentsList)`.
5. Return `? Construct(target, args, newTarget)`.

### 26.1.3 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)`.

### 26.1.4 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)`.

### 26.1.5 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. Let `receiver` be `target`.
4. Return `? target.[[Get]](key, receiver)`.

### 26.1.6 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)`.

### 26.1.7 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.

### 26.1.8 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)`.

### 26.1.9 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.
26.1.10 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).

26.1.11 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]]().

26.1.12 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. Let receiver be target.

26.1.13 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.

26.2 Proxy Objects

26.2.1 The Proxy Constructor
The Proxy constructor is the %Proxy% intrinsic object and the initial value of the Proxy property of the global object. When called as a constructor it creates and initializes a new proxy exotic object. Proxy is not intended to be called as a function and will throw an exception when called in that manner.

### 26.2.1.1 Proxy (target, handler)

When Proxy is called with arguments target and handler performs the following steps:

1. If NewTarget is undefined, throw a TypeError exception.

### 26.2.2 Properties of the Proxy Constructor

The value of the [[Prototype]] internal slot of the Proxy constructor is the intrinsic object %FunctionPrototype%.

The Proxy constructor does not have a prototype property because proxy exotic objects do not have a [[Prototype]] internal slot that requires initialization.

The Proxy constructor has the following properties:

### 26.2.2.1 Proxy.revocable (target, handler)

The Proxy.revocable function is used to create a revocable Proxy object. When Proxy.revocable is called with arguments target and handler, the following steps are taken:

1. Let p be ? ProxyCreate(target, handler).
2. Let revoker be a new built-in function object as defined in 26.2.2.1.1.
4. Let result be ObjectCreate(%ObjectPrototype%).
5. Perform CreateDataProperty(result, "proxy", p).
6. Perform CreateDataProperty(result, "revoke", revoker).
7. Return result.

### 26.2.2.1.1 Proxy Revocation Functions

A Proxy revocation function is an anonymous function that has the ability to invalidate a specific Proxy object.

Each Proxy revocation function has a [[RevableProxy]] internal slot.

When a Proxy revocation function, F, is called, the following steps are taken:
Let \( p \) be \( F.\{\text{RevocableProxy}\} \).

2. If \( p \) is \textbf{null}, return \textbf{undefined}.

3. Set \( F.\{\text{RevocableProxy}\} \) to \textbf{null}.

4. Assert: \( p \) is a Proxy object.

5. Set \( p.\{\text{ProxyTarget}\} \) to \textbf{null}.

6. Set \( p.\{\text{ProxyHandler}\} \) to \textbf{null}.

7. Return \textbf{undefined}.

The \texttt{length} property of a Proxy revocation function is 0.

### 26.3 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 \textit{ImportDeclaration} that includes a \textit{NameSpaceImport}.

In addition to the properties specified in 9.4.6 each Module Namespace Object has the following own property:

#### 26.3.1 @@toStringTag

The initial value of the \texttt{@@toStringTag} property is the String value \texttt{"Module"}.

This property has the attributes \{ [[Writable]]: \texttt{false}, [[Enumerable]]: \texttt{false}, [[Configurable]]: \texttt{false} \}.

### 27 Memory Model

The memory consistency model, or \textit{memory model}, specifies the possible orderings of \textit{Shared Data Block} events, arising via accessing \texttt{TypedArray} instances backed by a \texttt{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 \textit{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.

### 27.1 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>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Order]]</code></td>
<td>&quot;SeqCst&quot; or &quot;Unordered&quot;</td>
<td>The weakest ordering guaranteed by the memory model for the event.</td>
</tr>
<tr>
<td><code>[[NoTear]]</code></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><code>[[Block]]</code></td>
<td>A Shared Data Block</td>
<td>The block the event operates on.</td>
</tr>
<tr>
<td><code>[[ByteIndex]]</code></td>
<td>A nonnegative integer</td>
<td>The byte address of the read in <code>[[Block]]</code>.</td>
</tr>
<tr>
<td><code>[[ElementSize]]</code></td>
<td>A nonnegative integer</td>
<td>The size of the read.</td>
</tr>
</tbody>
</table>
Table 62: WriteSharedMemory Event Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Order]]</td>
<td>&quot;SeqCst&quot;, &quot;Unordered&quot;, or &quot;Init&quot;</td>
<td>The weakest ordering guaranteed by the memory model for the event.</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 nonnegative integer</td>
<td>The byte address of the write in [[Block]].</td>
</tr>
<tr>
<td>[[ElementSize]]</td>
<td>A nonnegative 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 63: ReadModifyWriteSharedMemory Event Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Order]]</td>
<td>&quot;SeqCst&quot;</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 nonnegative integer</td>
<td>The byte address of the read-modify-write in [[Block]].</td>
</tr>
<tr>
<td>[[ElementSize]]</td>
<td>A nonnegative 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 semantic function</td>
<td>A pure semantic function that returns a modified List of byte values from a 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.
In addition to Shared Data Block 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.

Shared Data Block events are ordered within candidate executions by the relations defined below.

### 27.2 Agent Events Records

An Agent Events Record is a Record with the following fields.

<table>
<thead>
<tr>
<th>Field</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>
</tbody>
</table>

### 27.3 Chosen Value Records

A Chosen Value Record is a Record with the following fields.

<table>
<thead>
<tr>
<th>Field</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>
### 27.4 Candidate Executions

A candidate execution of the evaluation of an agent cluster is a Record with the following fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[EventLists]]</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 semantic 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.

### 27.5 Abstract Operations for the Memory Model

#### 27.5.1 EventSet( execution )

The abstract operation EventSet takes one argument, a candidate execution execution. It performs the following steps:
1. Let *events* be an empty Set.
2. For each Agent Events Record *aer* in execution.[[EventLists]], do
   a. For each event *E* in *aer.[[EventList]], do
      i. Add *E* to *events*.
3. Return *events*.

### 27.5.2 SharedDataBlockEventSet( *execution* )

The abstract operation SharedDataBlockEventSet takes one argument, a candidate execution *execution*. It performs the following steps:

1. Let *events* be an empty Set.
2. For each event *E* in EventSet(*execution*), do
   a. If *E* is a ReadSharedMemory, WriteSharedMemory, or ReadModifyWriteSharedMemory event, add *E* to *events*.
3. Return *events*.

### 27.5.3 HostEventSet( *execution* )

The abstract operation HostEventSet takes one argument, a candidate execution *execution*. It performs the following steps:

1. Let *events* be an empty Set.
2. For each event *E* in EventSet(*execution*), do
   a. If *E* is not in SharedDataBlockEventSet(*execution*), add *E* to *events*.
3. Return *events*.

### 27.5.4 ComposeWriteEventBytes( *execution*, *byteIndex*, *Ws* )

The abstract operation ComposeWriteEventBytes takes four arguments, a candidate execution *execution*, a nonnegative integer *byteIndex*, and a List *Ws* of WriteSharedMemory or ReadModifyWriteSharedMemory events. It performs the following steps:

1. Let *byteLocation* be *byteIndex*.
2. Let *bytesRead* be a new empty List.
3. For each element *W* of *Ws* in List order, do
   a. Assert: *W* has *byteLocation* in its range.
   b. Let *payloadIndex* be *byteLocation* - *W.[[ByteIndex]]*.
   c. If *W* is a WriteSharedMemory event, then
      i. Let *byte* be *W.[[Payload]]*.*payloadIndex*.
   d. Else,
i. Assert: \( W \) is a \texttt{ReadModifyWriteSharedMemory} event.

ii. Let \texttt{bytes} be \texttt{ValueOfReadEvent}(\texttt{execution}, \texttt{W}).

iii. Let \texttt{bytesModified} be \( W.\texttt{[[ModifyOp]]}(\texttt{bytes}, \texttt{W.\texttt{[Payload]}]). \)

iv. Let \texttt{byte} be \texttt{bytesModified[payloadIndex]}. 

e. Append \texttt{byte} to \texttt{bytesRead}.

f. Increment \texttt{byteLocation} by 1.

4. Return \texttt{bytesRead}.

**NOTE 1**

The semantic function \( [[\text{ModifyOp}]] \) is given by the function properties on the Atomics object that introduce \texttt{ReadModifyWriteSharedMemory} events.

**NOTE 2**

This abstract operation composes a \texttt{List} of write events into a \texttt{List} of byte values. It is used in the event semantics of \texttt{ReadSharedMemory} and \texttt{ReadModifyWriteSharedMemory} events.

**27.5.5 ValueOfReadEvent( execution, \( R \) )**

The abstract operation \texttt{ValueOfReadEvent} takes two arguments, a candidate execution \texttt{execution} and a \texttt{ReadSharedMemory} or \texttt{ReadModifyWriteSharedMemory} event \( R \). It performs the following steps:

1. Assert: \( R \) is a \texttt{ReadSharedMemory} or \texttt{ReadModifyWriteSharedMemory} event.
2. Let \texttt{reads-bytes-from} be \texttt{execution.\texttt{[[ReadsBytesFrom]]}}.
3. Let \( W_s \) be \texttt{reads-bytes-from(R)}.
4. Assert: \( W_s \) is a \texttt{List} of \texttt{WriteSharedMemory} or \texttt{ReadModifyWriteSharedMemory} events with length equal to \( R.\texttt{[[ElementSize]]} \).
5. Return \texttt{ComposeWriteEventBytes}(\texttt{execution}, \( R.\texttt{[[ByteIndex]]}, W_s \)).

**27.6 Relations of Candidate Executions**

**27.6.1 agent-order**

For a candidate execution \texttt{execution}, \texttt{execution.\texttt{[[AgentOrder]]}} is a \texttt{Relation} that satisfies the following conditions.

1. Let \texttt{agent-order} be \texttt{execution.\texttt{[[AgentOrder]]}}.
2. For each pair of events \( E \) and \( D \) in \texttt{EventSet(execution)}), do
   a. For each \texttt{Agent Events Record aer} in \texttt{execution.\texttt{[[EventLists]]}}, do
      i. If \( E \) and \( D \) are in \texttt{aer.\texttt{[[EventList]]}} and \( E \) is before \( D \) in \texttt{List} order of \texttt{aer.\texttt{[[EventList]]}}, then \( E \) is \texttt{agent-order} before \( D \).

**NOTE**

Each \texttt{agent} introduces events in a per-agent strict total order during the evaluation. This is
27.6.2 reads-bytes-from

For a candidate execution execution, execution.\text{[[ReadsBytesFrom]]} is a semantic function from events in SharedDataBlockEventSet\text{(execution)} to Lists of events in SharedDataBlockEventSet\text{(execution)} that satisfies the following conditions.

1. Let reads-bytes-from be execution.\text{[[ReadsBytesFrom]]}.
2. For each ReadSharedMemory or ReadModifyWriteSharedMemory event \( R \) in SharedDataBlockEventSet\text{(execution)}, do
   a. There is a List of length equal to \( R.\text{[[ElementSize]]} \) of WriteSharedMemory or ReadModifyWriteSharedMemory events \( Ws \) such that reads-bytes-from\text{(R)} is \( Ws \).
   b. Let byteLocation be \( R.\text{[[ByteIndex]]} \).
   c. For each element \( W \) of \( Ws \) in List order, do
      i. \( W \) has byteLocation in its range.
      ii. \( W \) is not \( R \).
      iii. Increment byteLocation by 1.

27.6.3 reads-from

For a candidate execution execution, execution.\text{[[ReadsFrom]]} is the least Relation that satisfies the following conditions.

1. Let reads-from be execution.\text{[[ReadsFrom]]}.
2. Let reads-bytes-from be execution.\text{[[ReadsBytesFrom]]}.
3. For each pair of events \( R \) and \( W \) in SharedDataBlockEventSet\text{(execution)}, do
   a. Let \( Ws \) be reads-bytes-from\text{(R)}.
   b. Assert: \( Ws \) is a List of WriteSharedMemory or ReadModifyWriteSharedMemory events.
   c. If \( Ws \) contains \( W \), then \( R \) reads-from \( W \).

27.6.4 host-synchronizes-with

For a candidate execution execution, execution.\text{[[HostSynchronizesWith]]} is a host-provided strict partial order on host-specific events in HostEventSet\text{(execution)} that satisfies the following conditions.

1. Let host-synchronizes-with be execution.\text{[[HostSynchronizesWith]]}.
2. Let agent-order be execution.\text{[[AgentOrder]]}.
3. There is no cycle in the union of host-synchronizes-with and agent-order.

NOTE 1 For two host-specific events \( E \) and \( D \), \( E \) host-synchronizes-with \( D \) implies \( E \) happens-before
NOTE 2 The host-synchronizes-with relation allows the host to provide additional synchronization mechanisms, such as `postMessage` between HTML workers.

27.6.5 synchronizes-with

For a candidate execution `execution, execution`. `[[SynchronizesWith]]` is the least `Relation` that satisfies the following conditions.

1. Let `synchronizes-with` be `execution. [[SynchronizesWith]]`.
2. Let `reads-from` be `execution. [[ReadsFrom]]`.
3. Let `host-synchronizes-with` be `execution. [[HostSynchronizesWith]]`.
4. For each pair of events `R` and `W` in `SharedDataBlockEventSet(execution)` such that `R. [[Order]]` is "SeqCst" and `R reads-from W`, do
   a. Assert: `R` is a `ReadSharedMemory` or `ReadModifyWriteSharedMemory` event.
   b. Assert: `W` is a `WriteSharedMemory` or `ReadModifyWriteSharedMemory` event.
   c. If `W. [[Order]]` is "SeqCst" and `R` and `W` have equal ranges, then `W synchronizes-with R`.
   d. Else if `W` has `order "Init"`, then
      i. Let `allInitReads` be `true`.
      ii. For each event `V` such that `R reads-from V`, do
         1. If `V. [[Order]]` is not "Init", set `allInitReads` to `false`.
         iii. If `allInitReads` is `true`, then `W synchronizes-with R`.
5. For each pair of events `E` and `D` in `HostEventSet(execution)`, do
   a. If `E host-synchronizes-with D`, then `E synchronizes-with D`.

NOTE 1 Owing to convention, write events synchronizes-with read events, instead of read events synchronizes-with write events.

NOTE 2 Not all "SeqCst" events related by `reads-from` are related by synchronizes-with. Only events that also have equal ranges are related by synchronizes-with.

NOTE 3 For an event `R` and an event `W` such that `W synchronizes-with R, R` may `reads-from` other writes than `W`.

27.6.6 happens-before

For a candidate execution `execution, execution`. `[[HappensBefore]]` is the least `Relation` that satisfies the following conditions.

1. Let `happens-before` be `execution. [[HappensBefore]]`.
2. Let `agent-order` be `execution. [[AgentOrder]]`.
3. Let \texttt{synchronizes-with} be \textit{execution}.[[SynchronizesWith]].

4. For each pair of events \textit{E} and \textit{D} in \textit{EventSet}(\textit{execution}), do
   a. If \textit{E} is \textit{agent-order} before \textit{D}, then \textit{E happens-before} \textit{D}.
   b. If \textit{E} synchronizes-with \textit{D}, then \textit{E happens-before} \textit{D}.
   c. If \textit{E} and \textit{D} are in \textit{SharedDataBlockEventSet}(\textit{execution}), \textit{E}.[[Order]] is "\textbf{Init}", and \textit{E} and \textit{D} have overlapping ranges, then
      i. Assert: \textit{D}.[[Order]] is not "\textbf{Init}".
      ii. \textit{E happens-before} \textit{D}.
   d. If there is an event \textit{F} such that \textit{E happens-before} \textit{F} and \textit{F happens-before} \textit{D}, then \textit{E happens-before} \textit{D}.

\textbf{NOTE} Because happens-before is a superset of \textit{agent-order}, candidate executions are consistent with the single-thread evaluation semantics of ECMAScript.

### 27.7 Properties of Valid Executions

#### 27.7.1 Valid Chosen Reads

A candidate execution \textit{execution} has valid chosen reads if the following conditions hold.

1. For each \textit{ReadSharedMemory} or \textit{ReadModifyWriteSharedMemory} event \textit{R} in \textit{SharedDataBlockEventSet}(\textit{execution}), do
   a. Let \textit{chosenValue} be the element of \textit{execution}.[[ChosenValues]] whose \textit{[[Event]]} field is \textit{R}.
   b. Let \textit{readValue} be \textit{ValueOfReadEvent}(\textit{execution}, \textit{R}).
   c. Let \textit{chosenLen} be \textit{ValueOfReadEvent}(\textit{execution}, \textit{R}).
   d. Let \textit{readLen} be the number of elements of \textit{readValue}.
   e. \textit{chosenLen} is equal to \textit{readLen} and \textit{chosenValue}[i] is equal to \textit{readValue}[i] for all integer values \textit{i} in the range 0 through \textit{chosenLen}, exclusive.

#### 27.7.2 Coherent Reads

A candidate execution \textit{execution} has coherent reads if the following conditions hold.

1. Let \textit{happens-before} be \textit{execution}.[[HappensBefore]].
2. Let \textit{reads-bytes-from} be \textit{execution}.[[ReadsBytesFrom]].
3. For each \textit{ReadSharedMemory} or \textit{ReadModifyWriteSharedMemory} event \textit{R} in \textit{SharedDataBlockEventSet}(\textit{execution}), do
   a. Let \textit{Ws} be the List of events \textit{reads-bytes-from} (\textit{R}).
   b. Let \textit{byteLocation} be \textit{R}.[[ByteIndex]].
   c. For each element \textit{W} of \textit{Ws} in List order, do
i. It is not the case that \( R \) happens-before \( W \), and

ii. There is no \texttt{WriteSharedMemory} or \texttt{ReadModifyWriteSharedMemory} event \( V \) that has \textit{byteLocation} in its range such that \( W \) happens-before \( V \) and \( V \) happens-before \( R \).

iii. Increment \textit{byteLocation} by 1.

### 27.7.3 Tear Free Reads

A candidate execution \textit{execution} has tear free reads if the following conditions hold.

1. Let \texttt{reads-from} be \textit{execution}.[[ReadsFrom]].

2. For each \texttt{ReadSharedMemory} or \texttt{ReadModifyWriteSharedMemory} event \( R \) in \texttt{SharedDataBlockEventSet} (\textit{execution}), do
   a. If \( R.[[\text{NoTear}]] \) is \texttt{true}, then
      i. Assert: The remainder of dividing \( R.[[\text{ByteIndex}]] \) by \( R.[[\text{ElementSize}]] \) is 0.
      ii. For each event \( W \) such that \( R \) reads-from \( W \) and \( W.[[\text{NoTear}]] \) is \texttt{true}, do
         1. If \( R \) and \( W \) have equal ranges, then there is no \( V \) such that \( V \) and \( W \) have equal range, \( V.[[\text{NoTear}]] \) is \texttt{true}, \( W \) is not \( V \), and \( R \) reads-from \( V \).

\textbf{NOTE} An event's [[NoTear]] field is \texttt{true} when that event was introduced via accessing an integer TypedArray, and \texttt{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.

### 27.7.4 Sequentially Consistent Atomics

For a candidate execution \textit{execution}, memory-order is a strict total order of all events in \texttt{EventSet} (\textit{execution}) that satisfies the following conditions.

1. Let \texttt{happens-before} be \textit{execution}.[[HappensBefore]].

2. Let \texttt{synchronizes-with} be \textit{execution}.[[SynchronizesWith]].

3. For each pair of events \( E \) and \( D \) in \texttt{EventSet} (\textit{execution}), do
   a. If \( E \) happens-before \( D \), then \( E \) is memory-order before \( D \).
   b. If \( E \) and \( D \) are in \texttt{SharedDataBlockEventSet} (\textit{execution}) and \( E \) synchronizes-with \( D \), then
      i. Assert: \( D.[[\text{Order}]] \) is "\texttt{SeqCst}".
ii. There is no `WriteSharedMemory` or `ReadModifyWriteSharedMemory` event $W$ in $\text{SharedDataBlockEventSet}(\text{execution})$ with equal range as $D$ such that $W$ is not $E$, $E$ is memory-order before $W$, and $W$ is memory-order before $D$.

iii. NOTE: This clause additionally constrains "SeqCst" events on equal ranges.

4. For each `WriteSharedMemory` or `ReadModifyWriteSharedMemory` event $W$ in $\text{SharedDataBlockEventSet}(\text{execution})$, do
   a. If $W.([\text{Order}])$ is "SeqCst", then it is not the case that there is an infinite number of $\text{ReadSharedMemory}$ or $\text{ReadModifyWriteSharedMemory}$ events in $\text{SharedDataBlockEventSet}(\text{execution})$ with equal range that is memory-order before $W$.
   b. NOTE: 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 | While memory-order includes all events in $\text{EventSet}(\text{execution})$, those that are not constrained by happens-before or synchronizes-with are allowed to occur anywhere in the order. |

### 27.7.5 Valid Executions

A candidate execution $\text{execution}$ is a valid execution (or simply an execution) if the following conditions hold.

1. The host provides a host-synchronizes-with Relation for $\text{execution}.[[\text{HostSynchronizesWith}]]$, and
2. $\text{execution}.[[\text{HappensBefore}]]$ is a strict partial order, and
3. $\text{execution}$ has valid chosen reads, and
4. $\text{execution}$ has coherent reads, and
5. $\text{execution}$ has tear free reads, and
6. $\text{execution}$ has sequentially consistent atomics.

All programs have at least one valid execution.

### 27.8 Races

For an execution $\text{execution}$, two events $E$ and $D$ in $\text{SharedDataBlockEventSet}(\text{execution})$ are in a race if the following conditions hold.

1. Let happens-before be $\text{execution}.[[\text{HappensBefore}]]$.
2. Let reads-from be $\text{execution}.[[\text{ReadsFrom}]]$.
3. $E$ is not $D$, and
4. It is not the case that $E$ happens-before $D$ or $D$ happens-before $E$, and
5. If $E$ and $D$ are both WriteSharedMemory or ReadModifyWriteSharedMemory events, then
   a. $E$ and $D$ do not have disjoint ranges.
6. Else,
   a. $E$ reads-from $D$ or $D$ reads-from $E$.

27.9 Data Races

For an execution execution, two events $E$ and $D$ in SharedDataBlockEventSet(execution) are in a data race if the following conditions hold.

1. $E$ and $D$ are in a race in execution, and
2. At least one of $E$ or $D$ does not have [[Order]] "SeqCst" or $E$ and $D$ have overlapping ranges.

27.10 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.

27.11 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.

- **Atoms are carved in stone**: Program transformations must not cause the "SeqCst" events in an agent-order slice to be reordered with its "Unordered" operations, nor its "SeqCst" operations to be reordered with each other, nor may a program transformation remove a "SeqCst" operation from the agent-order.

  (In practice, the prohibition on reorderings forces a compiler to assume that every "SeqCst" operation is a synchronization and included in the final memory-order, which it would usually have to assume anyway in the absence of inter-agent program analysis. It also forces the compiler to assume that every call where the callee's effects on the memory-order are unknown may contain "SeqCst" operations.)

- ** Reads must be stable**: Any given shared memory read must only observe a single value in an execution.

  (For example, if what is semantically a single read in the program is executed multiple times then the program is subsequently allowed to observe only one of the values read. A transformation known as rematerialization can violate this rule.)

- ** Writes must be stable**: All observable writes to shared memory must follow from program semantics in an execution.
(For example, a transformation may not introduce certain observable writes, such as by using read-modify-write operations on a larger location to write a smaller datum, writing a value to memory that the program could not have written, or writing a just-read value back to the location it was read from, if that location could have been overwritten by another agent after the read.)

- **Possible read values must be nonempty**: Program transformations cannot cause the possible read values of a shared memory read to become empty.

(Counterintuitively, this rule in effect restricts transformations on writes, because writes have force in memory model insofar as to be read by read events. For example, writes may be moved and coalesced and sometimes reordered between two "SeqCst" operations, but the transformation may not remove every write that updates a location; some write must be preserved.)

Examples of transformations that remain valid are: merging multiple non-atomic reads from the same location, reordering non-atomic reads, introducing speculative non-atomic reads, merging multiple non-atomic writes to the same location, reordering non-atomic writes to different locations, and hoisting non-atomic reads out of loops even if that affects termination. Note in general that aliased TypedArrays make it hard to prove that locations are different.

**NOTE 3**

The following are guidelines for ECMAScript implementers generating machine code for shared memory accesses.

For architectures with memory models no weaker than those of ARM or Power, non-atomic stores and loads may be compiled to bare stores and loads on the target architecture. Atomic stores and loads may be compiled down to instructions that guarantee sequential consistency. If no such instructions exist, memory barriers are to be employed, such as placing barriers on both sides of a bare store or load. Read-modify-write operations may be compiled to read-modify-write instructions on the target architecture, such as LOCK-prefixed instructions on x86, load-exclusive/store-exclusive instructions on ARM, and load-link/store-conditional instructions on Power.

Specifically, the memory model is intended to allow code generation as follows:

- Every atomic operation in the program is assumed to be necessary.
- Atomic operations are never rearranged with each other or with non-atomic operations.
- Functions are always assumed to perform atomic operations.
- Atomic operations are never implemented as read-modify-write operations on larger data, but as non-lock-free atomics if the platform does not have atomic operations of
the appropriate size. (We already assume that every platform has normal memory access operations of every interesting size.)

Naive code generation uses these patterns:

- Regular loads and stores compile to single load and store instructions.
- Lock-free atomic loads and stores compile to a full (sequentially consistent) fence, a regular load or store, and a full fence.
- Lock-free atomic read-modify-write accesses compile to a full fence, an atomic read-modify-write instruction sequence, and a full fence.
- Non-lock-free atomics compile to a spinlock acquire, a full fence, a series of non-atomic load and store instructions, a full fence, and a spinlock release.

That mapping is correct so long as an atomic operation on an address range does not race with a non-atomic write or with an atomic operation of different size. However, that is all we need: the memory model effectively demotes the atomic operations involved in a race to non-atomic status. On the other hand, the naive mapping is quite strong: it allows atomic operations to be used as sequentially consistent fences, which the memory model does not actually guarantee.

A number of local improvements to those basic patterns are also intended to be legal:

- There are obvious platform-dependent improvements that remove redundant fences. For example, on x86 the fences around lock-free atomic loads and stores can always be omitted except for the fence following a store, and no fence is needed for lock-free read-modify-write instructions, as these all use LOCK-prefixed instructions. On many platforms there are fences of several strengths, and weaker fences can be used in certain contexts without destroying sequential consistency.
- Most modern platforms support lock-free atomics for all the data sizes required by ECMAScript atomics. Should non-lock-free atomics be needed, the fences surrounding the body of the atomic operation can usually be folded into the lock and unlock steps. The simplest solution for non-lock-free atomics is to have a single lock word per SharedArrayBuffer.
- There are also more complicated platform-dependent local improvements, requiring some code analysis. For example, two back-to-back fences often have the same effect as a single fence, so if code is generated for two atomic operations in sequence, only a single fence need separate them. On x86, even a single fence separating atomic stores can be omitted, as the fence following a store is only needed to separate the store from a subsequent load.
A Grammar Summary

A.1 Lexical Grammar

SourceCharacter ::
   any Unicode code point

InputElementDiv ::
   WhiteSpace
   LineTerminator
   Comment
   CommonToken
   DivPunctuator
   RightBracePunctuator

InputElementRegExp ::
   WhiteSpace
   LineTerminator
   Comment
   CommonToken
   RightBracePunctuator
   RegularExpressionLiteral

InputElementRegExpOrTemplateTail ::
   WhiteSpace
   LineTerminator
   Comment
   CommonToken
   RegularExpressionLiteral
   TemplateSubstitutionTail

InputElementTemplateTail ::
   WhiteSpace
   LineTerminator
   Comment
   CommonToken
   DivPunctuator
   TemplateSubstitutionTail

WhiteSpace ::
   <TAB>
LineTerminator ::
   <LF>
   <CR>
   <LS>
   <PS>

LineTerminatorSequence ::
   <LF>
   <CR>[lookahead ≠ <LF>]
   <LS>
   <PS>
   <CR><LF>

Comment ::
   MultiLineComment
   SingleLineComment

MultiLineComment ::
   /* MultiLineCommentChars_{opt} */

MultiLineCommentChars ::
   MultiLineNotAsteriskChar MultiLineCommentChars_{opt}
   * PostAsteriskCommentChars_{opt}

PostAsteriskCommentChars ::
   MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars_{opt}
   * PostAsteriskCommentChars_{opt}

MultiLineNotAsteriskChar ::
   SourceCharacter but not *

MultiLineNotForwardSlashOrAsteriskChar ::
   SourceCharacter but not one of / or *

SingleLineComment ::
   // SingleLineCommentChars_{opt}
SingleLineCommentChars ::
    SingleLineCommentChar SingleLineCommentChars opt

SingleLineCommentChar ::
    SourceCharacter but not LineTerminator

CommonToken ::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral
    Template

IdentifierName ::
    IdentifierStart
    IdentifierName IdentifierPart

IdentifierStart ::
    UnicodeIDStart
    $
    _
    \\ UnicodeEscapeSequence

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

UnicodeIDStart ::
    any Unicode code point with the Unicode property &ldquo;ID_Start&rdquo;

UnicodeIDContinue ::
    any Unicode code point with the Unicode property &ldquo;ID_Continue&rdquo;

ReservedWord ::
    Keyword
    FutureReservedWord
    NullLiteral
    BooleanLiteral
**Keyword ::**

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

**FutureReservedWord ::**

```plaintext
enum
```

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

The following tokens are also considered to be `FutureReservedWords` when parsing strict mode code:

```plaintext
implements
package
protected
interface
private
public
```

**Punctuator ::**

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

RightBracePunctuator ::
    }

NullLiteral ::
    null

BooleanLiteral ::
    true
    false

NumericLiteral ::
    DecimalLiteral
    BinaryIntegerLiteral
    OctalIntegerLiteral
    HexIntegerLiteral

DecimalLiteral ::
    DecimalIntegerLiteral . DecimalDigits opt . ExponentPart opt
    . DecimalDigits ExponentPart opt
    DecimalIntegerLiteral ExponentPart opt

DecimalIntegerLiteral ::
    0
    NonZeroDigit DecimalDigits opt

DecimalDigits ::
    DecimalDigit
    DecimalDigits DecimalDigit

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

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

ExponentPart ::
    ExponentIndicator SignedInteger

ExponentIndicator :: one of
    e E
SignedInteger ::
    DecimalDigits
    + DecimalDigits
    - DecimalDigits

BinaryIntegerLiteral ::
    \0b BinaryDigits
    \0B BinaryDigits

BinaryDigits ::
    BinaryDigit
    BinaryDigits BinaryDigit

BinaryDigit :: one of
    0 1

OctalIntegerLiteral ::
    \0o OctalDigits
    \0O OctalDigits

OctalDigits ::
    OctalDigit
    OctalDigits OctalDigit

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

HexIntegerLiteral ::
    \0x HexDigits
    \0X HexDigits

HexDigits ::
    HexDigit
    HexDigits HexDigit

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

StringLiteral ::
    " DoubleStringCharacters\opt "
    ' SingleStringCharacters\opt '

DoubleStringCharacters ::
    DoubleStringCharacter DoubleStringCharacters\opt
SingleStringCharacters ::
  SingleStringCharacter SingleStringCharacters opt

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

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

LineContinuation ::
  \ LineTerminatorSequence

EscapeSequence ::
  CharacterEscapeSequence
  \ [lookahead ∈ DecimalDigit]
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence ::
  SingleEscapeCharacter
  NonEscapeCharacter

SingleEscapeCharacter :: one of
  ' " \ b f n r t v

NonEscapeCharacter ::
  SourceCharacter but not one of EscapeCharacter or LineTerminator

EscapeCharacter ::
  SingleEscapeCharacter
  DecimalDigit
  x
  u

HexEscapeSequence ::
  x HexDigit HexDigit

UnicodeEscapeSequence ::
  u Hex4Digits
  u{ HexDigits }
Hex4Digits ::
   HexDigit HexDigit HexDigit HexDigit

RegularExpressionLiteral ::
   / RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody ::
   RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars ::
   [empty]
   RegularExpressionChars RegularExpressionChar

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

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

RegularExpressionBackslashSequence ::
   \ RegularExpressionNonTerminator

RegularExpressionNonTerminator ::
   SourceCharacter but not LineTerminator

RegularExpressionClass ::
   [ RegularExpressionClassChars ]

RegularExpressionClassChars ::
   [empty]
   RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar ::
   RegularExpressionNonTerminator but not one of ] or \ 
   RegularExpressionBackslashSequence

RegularExpressionFlags ::
   [empty]
   RegularExpressionFlags IdentifierPart
Template ::
  NoSubstitutionTemplate
  TemplateHead

NoSubstitutionTemplate ::
  ` TemplateCharacters_opt `

TemplateHead ::
  ` TemplateCharacters_opt ${

TemplateSubstitutionTail ::
  TemplateMiddle
  TemplateTail

TemplateMiddle ::
  } TemplateCharacters_opt ${

TemplateTail ::
  } TemplateCharacters_opt `

TemplateCharacters ::
  TemplateCharacter TemplateCharacters_opt

TemplateCharacter ::
  $ [lookahead ≠ {]
  \ EscapeSequence
  LineContinuation
  Line TerminatorSequence
  SourceCharacter but not one of ` or \ or $ or Line Terminator

A.2 Expressions

IdentifierReference[Yield, Await] :
  Identifier
  [ ~Yield] yield
  [ ~Await] await

BindingIdentifier[Yield, Await] :
  Identifier
  [ ~Yield] yield
  [ ~Await] await
Identifier:
   IdentifierName but not ReservedWord

AsyncArrowBindingIdentifier[Yield] :
   BindingIdentifier[?Yield, +Await]

LabelIdentifier[Yield, Await] :
   Identifier
   [~Yield] yield
   [~Await] await

PrimaryExpression[Yield, Await] :
   this
   IdentifierReference[?Yield, ?Await]
   Literal
   ArrayLiteral[?Yield, ?Await]
   ObjectLiteral[?Yield, ?Await]
   FunctionExpression
   ClassExpression[?Yield, ?Await]
   GeneratorExpression
   AsyncFunctionExpression
   RegularExpressionLiteral
   TemplateLiteral[?Yield, ?Await]
   CoverParenthesizedExpressionAndArrowParameterList[?Yield, ?Await]

CoverParenthesizedExpressionAndArrowParameterList[Yield, Await] :
   ( Expression[+In, ?Yield, ?Await] )
   ( Expression[+In, ?Yield, ?Await] , )
   ( )
   ( ... BindingIdentifier[?Yield, ?Await] )
   ( ... BindingPattern[?Yield, ?Await] )
   ( Expression[+In, ?Yield, ?Await] , ... BindingIdentifier[?Yield, ?Await] )
   ( Expression[+In, ?Yield, ?Await] , ... BindingPattern[?Yield, ?Await] )

When processing an instance of the production PrimaryExpression :
CoverParenthesizedExpressionAndArrowParameterList the interpretation of
CoverParenthesizedExpressionAndArrowParameterList is refined using the following grammar:

ParenthesizedExpression[Yield, Await] :
   ( Expression[+In, ?Yield, ?Await] )
Literal:
- NullLiteral
- BooleanLiteral
- NumericLiteral
- StringLiteral

ArrayLiteral[Yield, Await] :
  [ Elision_opt ]
  [ 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]

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]
  MethodDefinition[?Yield, ?Await]
Property

Yield, Await

Literal

Computed

Yield, Await

Literal

Identifier

String

Numeric

Computed

Yield, Await

Assignment

[ +In, Yield, Await ]

Cover

Identifier

Yield, Await

Initializer

In, Yield, Await

= Assignment

[ ?In, Yield, Await ]

Template

Yield, Await

NoSubstitution

Expression

[ +In, Yield, Await ]

Template

Yield, Await

Tail

Template

Yield, Await

MiddleList

Template

Yield, Await

Middle

Expression

[ +In, Yield, Await ]

Member

Yield, Await

Expression

[ ?Yield, Await ]

Member

Yield, Await

Expression

[ +In, Yield, Await ]

Identifier

Member

Yield, Await

Template

Yield, Await

Literal

Template

Yield, Await

Super

Yield, Await

Super

Yield, Await

Expression

[ ?Yield, Await ]

Arguments

Yield, Await

super

Expression

[ +In, Yield, Await ]

Arguments

Yield, Await

super

Expression

[ +In, Yield, Await ]
super . IdentifierName

MetaProperty:
NewTarget

NewTarget:

    new . target

NewExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await]
    new NewExpression[?Yield, ?Await]

CallExpression[Yield, Await] :
    CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await]
    SuperCall[?Yield, ?Await]
    CallExpression[?Yield, ?Await] [ Expression[+In, ?Yield, ?Await] ]
    CallExpression[?Yield, ?Await] . IdentifierName

CoverCallExpressionAndAsyncArrowHead[Yield, Await] :
    MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]

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

CallMemberExpression[Yield, Await] :
    MemberExpression[?Yield, ?Await] Arguments[?Yield, ?Await]

SuperCall[Yield, Await] :
    super Arguments[?Yield, ?Await]

Arguments[Yield, Await] :
    ( )
    ( ArgumentList[?Yield, ?Await] , )

ArgumentList[Yield, Await] :
    AssignmentExpression[+In, ?Yield, ?Await]
    ... AssignmentExpression[+In, ?Yield, ?Await]
ArgumentList[?Yield, ?Await] , ... AssignmentExpression[+In, ?Yield, ?Await]

LeftHandSideExpression[Yield, Await] :
NewExpression[?Yield, ?Await]
CallExpression[?Yield, ?Await]

UpdateExpression[Yield, Await] :
LeftHandSideExpression[?Yield, ?Await]
LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] ++
LeftHandSideExpression[?Yield, ?Await] [no LineTerminator here] --
++ UnaryExpression[?Yield, ?Await]
-- UnaryExpression[?Yield, ?Await]

UnaryExpression[Yield, Await] :
UpdateExpression[?Yield, ?Await]
delete UnaryExpression[?Yield, ?Await]
void UnaryExpression[?Yield, ?Await]
typeof UnaryExpression[?Yield, ?Await]
+ UnaryExpression[?Yield, ?Await]
- UnaryExpression[?Yield, ?Await]
~ UnaryExpression[?Yield, ?Await]
! UnaryExpression[?Yield, ?Await]
[+Await] AwaitExpression[?Yield]

ExponentiationExpression[Yield, Await] :
UnaryExpression[?Yield, ?Await]

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

MultiplicativeOperator : one of
* / %

AdditiveExpression[Yield, Await] :
MultiplicativeExpression[?Yield, ?Await]

ShiftExpression[Yield, Await] :
  AdditiveExpression[?Yield, ?Await]

RelationalExpression[In, Yield, Await] :
  ShiftExpression[?Yield, ?Await]
  RelationalExpression[?In, ?Yield, ?Await] instanceof
    ShiftExpression[?Yield, ?Await]

EqualityExpression[In, Yield, Await] :
  RelationalExpression[?In, ?Yield, ?Await]
  EqualityExpression[?In, ?Yield, ?Await] ===
    RelationalExpression[?In, ?Yield, ?Await]
  EqualityExpression[?In, ?Yield, ?Await] !==
    RelationalExpression[?In, ?Yield, ?Await]

BitwiseANDExpression[In, Yield, Await] :
  EqualityExpression[?In, ?Yield, ?Await]

BitwiseXORExpression[In, Yield, Await] :
  BitwiseANDExpression[?In, ?Yield, ?Await]
  BitwiseXORExpression[?In, ?Yield, ?Await] ^
    BitwiseANDExpression[?In, ?Yield, ?Await]

BitwiseORExpression[In, Yield, Await] :
  BitwiseXORExpression[?In, ?Yield, ?Await]

LogicalANDExpression[In, Yield, Await] :
BitwiseORExpression[?In, ?Yield, ?Await] &&
BitwiseORExpression[?In, ?Yield, ?Await]

LogicalORExpression[In, Yield, Await] :
LogicalANDExpression[?In, ?Yield, ?Await]
LogicalORExpression[?In, ?Yield, ?Await] |

ConditionalExpression[In, Yield, Await] :
LogicalORExpression[?In, ?Yield, ?Await]
LogicalORExpression[?In, ?Yield, ?Await] ?
AssignmentExpression[+In, ?Yield, ?Await] :
AssignmentExpression[?In, ?Yield, ?Await]

AssignmentExpression[In, Yield, Await] :
ConditionalExpression[?In, ?Yield, ?Await]
[+Yield] YieldExpression[?In, ?Await]
ArrowFunction[?In, ?Yield, ?Await]
AsyncArrowFunction[?In, ?Yield, ?Await]
LeftHandSideExpression[?Yield, ?Await] AssignmentOperator
AssignmentExpression[?In, ?Yield, ?Await]

In certain circumstances when processing an instance of the production AssignmentExpression : LeftHandSideExpression = AssignmentExpression the following grammar is used to refine the interpretation of LeftHandSideExpression:

AssignmentPattern[Yield, Await] :
ObjectAssignmentPattern[?Yield, ?Await]
ArrayAssignmentPattern[?Yield, ?Await]

ObjectAssignmentPattern[Yield, Await] :
{
  
  { AssignmentPropertyList[?Yield, ?Await] }
  
  { AssignmentPropertyList[?Yield, ?Await] , }
}
ArrayAssignmentPattern[Yield, Await] :
  [ Elision \text{opt} AssignmentRestElement[?Yield, ?Await] \text{opt} ]
  [ AssignmentElementList[?Yield, ?Await], Elision \text{opt}
    AssignmentRestElement[?Yield, ?Await] \text{opt} ]

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 \text{opt} AssignmentElement[?Yield, ?Await]

AssignmentProperty[Yield, Await] :
  IdentifierReference[?Yield, ?Await] \text{Initializer}[+In, ?Yield, ?Await] \text{opt}
 (PropertyName[?Yield, ?Await] : AssignmentElement[?Yield, ?Await]

AssignmentElement[Yield, Await] :
  DestructuringAssignmentTarget[?Yield, ?Await] \text{Initializer}[+In, ?Yield, ?Await] \text{opt}

AssignmentRestElement[Yield, Await] :
  ... DestructuringAssignmentTarget[?Yield, ?Await]

DestructuringAssignmentTarget[Yield, Await] :
  LeftHandSideExpression[?Yield, ?Await]

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

Expression[In, Yield, Await] :
  AssignmentExpression[?In, ?Yield, ?Await]
  Expression[?In, ?Yield, ?Await], AssignmentExpression[?In, ?Yield, ?Await]

A.3 Statements
Statement[Yield, Await, Return] :
  BlockStatement[?Yield, ?Await, ?Return]
  VariableStatement[?Yield, ?Await]
  EmptyStatement
  ExpressionStatement[?Yield, ?Await]
  IfStatement[?Yield, ?Await, ?Return]
  ContinueStatement[?Yield, ?Await]
  BreakStatement[?Yield, ?Await]
  [?Return] ReturnStatement[?Yield, ?Await]
  WithStatement[?Yield, ?Await, ?Return]
  LabelledStatement[?Yield, ?Await, ?Return]
  ThrowStatement[?Yield, ?Await]
  TryStatement[?Yield, ?Await]
  DebuggerStatement

Declaration[Yield, Await] :
  HoistableDeclaration[?Yield, ?Await, ~Default]
  ClassDeclaration[?Yield, ?Await, ~Default]
  LexicalDeclaration[+In, ?Yield, ?Await]

HoistableDeclaration[Yield, Await, Default] :
  FunctionDeclaration[?Yield, ?Await, ?Default]
  GeneratorDeclaration[?Yield, ?Await, ?Default]
  AsyncFunctionDeclaration[?Yield, ?Await, ?Default]

BreakableStatement[Yield, Await, Return] :
  IterationStatement[?Yield, ?Await, ?Return]
  SwitchStatement[?Yield, ?Await, ?Return]

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

Block[Yield, Await, Return] :

StatementList[Yield, Await, Return] :
  StatementListListItem[?Yield, ?Await, ?Return]
StatementListItem[Yield, Await, Return] :
    Statement[?Yield, ?Await, ?Return]
    Declaration[?Yield, ?Await]

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

LetOrConst :
    let
cast

BindingList[In, Yield, Await] :
    LexicalBinding[?In, ?Yield, ?Await]
    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]
    VariableDeclarationList[?In, ?Yield, ?Await],
    VariableDeclaration[?In, ?Yield, ?Await]

VariableDeclaration[In, Yield, Await] :

BindingPattern[Yield, Await] :
    ObjectBindingPattern[?Yield, ?Await]
    ArrayBindingPattern[?Yield, ?Await]

ObjectBindingPattern[Yield, Await] :
    { }
    { BindingPropertyList[?Yield, ?Await] }
    { BindingPropertyList[?Yield, ?Await], }

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

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]

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

SimpleNameBinding[?Yield, Await] :

BindingRestElement[?Yield, Await] :
  ... BindingIdentifier[?Yield, ?Await]
  ... BindingPattern[?Yield, ?Await]

EmptyStatement :
  ;

ExpressionStatement[?Yield, ?Await] :

IfStatement[?Yield, Await, Return] :

IterationStatement[Yield, Await, Return] :
    Statement[?Yield, ?Await, ?Return]
  for ( var VariableDeclarationList[^In, ?Yield, ?Await] ;
    Statement[?Yield, ?Await, ?Return]
  for ( [lookahead \notin {let []}] LeftHandSideExpression[^Yield, ?Await] in
    Statement[?Yield, ?Await, ?Await]
  for ( ForDeclaration[^Yield, ?Await] in Expression[^In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
  for ( [lookahead \neq let] LeftHandSideExpression[^Yield, ?Await] of
    AssignmentExpression[^In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]
    Statement[?Yield, ?Await, ?Return]
  for ( ForDeclaration[^Yield, ?Await] of AssignmentExpression[^In, ?Yield, ?Await] )
    Statement[?Yield, ?Await, ?Return]

ForDeclaration[Yield, Await] :
  LetOrConst ForBinding[^Yield, ?Await]

ForBinding[Yield, Await] :
  BindingIdentifier[^Yield, ?Await]
  BindingPattern[^Yield, ?Await]

ContinueStatement[Yield, Await] :
  continue ;

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] opt }

CaseClauses[Yield, Await, Return] :
  CaseClause[?Yield, ?Await, ?Return]

CaseClause[Yield, Await, Return] :

DefaultClause[Yield, Await, Return] :

LabelledStatement[Yield, Await, Return] :

LabelledItem[Yield, Await, Return] :
  Statement[?Yield, ?Await, ?Return]
  FunctionDeclaration[?Yield, ?Await, ~Default]

ThrowStatement[Yield, Await] :

TryStatement[Yield, Await, Return] :
Finally[?Yield, ?Await, ?Return]

Catch[?Yield, ?Await, ?Return] :

Finally[?Yield, ?Await, ?Return] :
finally Block[?Yield, ?Await, ?Return]

CatchParameter[?Yield, ?Await] :
BindingIdentifier[?Yield, ?Await]
BindingPattern[?Yield, ?Await]

DebuggerStatement :
debugger ;

A.4 Functions and Classes

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] }

UniqueFormalParameters[?Yield, ?Await] :
FormalParameters[?Yield, ?Await]

FormalParameters[?Yield, ?Await] :
[empty]
FunctionRestParameter[?Yield, ?Await]
FormalParameterList[?Yield, ?Await]
FormalParameterList[?Yield, ?Await] ,

FormalParameterList[?Yield, ?Await] :
FormalParameter[?Yield, ?Await]

FunctionRestParameter[?Yield, Await] :
BindingRestElement[?Yield, ?Await]

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

FunctionBody[?Yield, Await] :
FunctionStatementList[?Yield, ?Await]

FunctionStatementList[?Yield, Await] :

ArrowFunction[In, Yield, Await] :
ArrowParameters[?Yield, ?Await] [no LineTerminator here] => ConciseBody[?In]

ArrowParameters[?Yield, Await] :
BindingIdentifier[?Yield, ?Await]
CoverParenthesizedExpressionAndArrowParameterList[?Yield, ?Await]

ConciseBody[?In] :
[lookahead ≠ {} AssignmentExpression[?In, ~Yield, ~Await]
{} FunctionBody[~Yield, ~Await] ]

When the production ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList is recognized the following grammar is used to refine the interpretation of CoverParenthesizedExpressionAndArrowParameterList:

ArrowFormalParameters[?Yield, ?Await] :
( UniqueFormalParameters[?Yield, ?Await] )

AsyncArrowFunction[In, Yield, Await] :
CoverCallExpressionAndAsyncArrowHead[?Yield, ?Await] [no LineTerminator here] =>
AsyncConciseBody[?In]

AsyncConciseBody[?In] :
[lookahead ≠ {} AssignmentExpression[?In, ~Yield, +Await]
When the production `AsyncArrowFunction : CoverCallExpressionAndAsyncArrowHead [no LineTerminator here] => AsyncConciseBody` is recognized the following grammar is used to refine the interpretation of `CoverParenthesizedExpressionAndArrowParameterList`:

`AsyncArrowHead`:

```latex
dot:\text{async} \ [\text{no LineTerminator here}] \ ArrowFormalParameters[^{+Yield, +Await}] \end{verbatim}
```

`MethodDefinition[^{Yield, Await}]`:

```latex
\text{PropertyName}[^{+Yield, ?Await}] ( \text{UniqueFormalParameters}[^{~Yield, ~Await}] ) \{ \text{FunctionBody}[^{~Yield, ~Await}] \}
```

`GeneratorMethod[^{+Yield, ?Await}]`:

```latex
\text{get} \text{PropertyName}[^{+Yield, ?Await}] ( ) \{ \text{FunctionBody}[^{~Yield, ~Await}] \}
```

```latex
\text{set} \text{PropertyName}[^{+Yield, ?Await}] ( \text{PropertySetParameterList} ) \{ \text{FunctionBody}[^{~Yield, ~Await}] \}
```

`PropertySetParameterList`:

```latex
\text{FormalParameter}[^{~Yield, ~Await}] \end{verbatim}
```

`GeneratorMethod[^{Yield, Await}]`:

```latex
* \text{PropertyName}[^{+Yield, ?Await}] ( \text{UniqueFormalParameters}[^{+Yield, ~Await}] ) \{ \text{GeneratorBody} \}
```

`GeneratorDeclaration[^{Yield, Await, Default}]`:

```latex
\text{function} * \text{BindingIdentifier}[^{+Yield, ?Await}] ( \text{FormalParameters}[^{+Yield, ~Await}] ) \{ \text{GeneratorBody} \}
```

```latex
[^{Default}] \text{function} * ( \text{FormalParameters}[^{+Yield, ~Await}] ) \{ \text{GeneratorBody} \}
```

`GeneratorExpression`:

```latex
\text{function} * \text{BindingIdentifier}[^{+Yield, ~Await}] \text{opt} ( \text{FormalParameters}[^{+Yield, ~Await}] ) \{ \text{GeneratorBody} \}
```

`GeneratorBody`:

```latex
\text{FunctionBody}[^{+Yield, ~Await}] \end{verbatim}
```

`YieldExpression[^{In, Await}]`:

```latex
\text{yield} \end{verbatim}
```

```latex
\text{yield} \ [\text{no LineTerminator here}] \text{AssignmentExpression}[^{?In, +Yield, ?Await}] \end{verbatim}
```
yield [no LineTerminator here] * AssignmentExpression[?In, +Yield, ?Await]

AsyncMethod[ Yield, Await ] :

AsyncFunctionDeclaration[ Yield, Await, Default ] :

AsyncFunctionExpression :
  async [no LineTerminator here] function ( FormalParameters[~Yield, +Await] ) { AsyncFunctionBody }

AsyncFunctionBody :
  FunctionBody[~Yield, +Await]

AwaitExpression[ Yield ] :
  await UnaryExpression[?Yield, +Await]

ClassDeclaration[ Yield, Await, Default ] :
  class BindingIdentifier[?Yield, ?Await] ClassTail[?Yield, ?Await]
  [+Default] class ClassTail[?Yield, ?Await]

ClassExpression[ Yield, Await ] :
  class BindingIdentifier[?Yield, ?Await] opt ClassTail[?Yield, ?Await]

ClassTail[ Yield, Await ] :
  ClassHeritage[?Yield, ?Await] opt { ClassBody[?Yield, ?Await] opt }

ClassHeritage[ Yield, Await ] :
  extends LeftHandSideExpression[?Yield, ?Await]

ClassBody[ Yield, Await ] :
  ClassElementList[?Yield, ?Await]

ClassElementList[ Yield, Await ] :
  ClassElement[?Yield, ?Await]
A.5 Scripts and Modules

Script :
  
  ScriptBody_{opt}

ScriptBody :
  
  StatementList[{~Yield, ~Await, ~Return}]

Module :
  
  ModuleBody_{opt}

ModuleBody :
  
  ModuleItemList

ModuleItemList :
  
  ModuleItem
  
  ModuleItemList ModuleItem

ModuleItem :
  
  ImportDeclaration
  ExportDeclaration
  StatementListItem[{~Yield, ~Await, ~Return}]

ImportDeclaration :
  
  import ImportClause FromClause ;
  import ModuleSpecifier ;

ImportClause :
  
  ImportedDefaultBinding
  NameSpaceImport
  NamedImports
  ImportedDefaultBinding , NameSpaceImport
  ImportedDefaultBinding , NamedImports

ImportedDefaultBinding :
ImportedBinding

NameSpaceImport:
    * as ImportedBinding

NamedImports:
    { }
    { ImportsList }
    { ImportsList , }

FromClause:
    from ModuleSpecifier

ImportsList:
    ImportSpecifier
    ImportsList , ImportSpecifier

ImportSpecifier:
    ImportedBinding
    IdentifierName as ImportedBinding

ModuleSpecifier:
    StringLiteral

ImportedBinding:
    BindingIdentifier[~Yield, ~Await]

ExportDeclaration:
    export * FromClause ;
    export ExportClause FromClause ;
    export ExportClause ;
    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] ;

ExportClause:
    { }
    { ExportsList }
    { ExportsList , }
ExportsList : 
  ExportSpecifier 
  ExportsList , ExportSpecifier

ExportSpecifier : 
  IdentifierName 
  IdentifierName as IdentifierName

### A.6 Number Conversions

StringNumericLiteral :::
  StrWhiteSpaceopt 
  StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt

StrWhiteSpace :::
  StrWhiteSpaceChar StrWhiteSpaceopt

StrWhiteSpaceChar :::
  WhiteSpace 
  LineTerminator

StrNumericLiteral :::
  StrDecimalLiteral 
  BinaryIntegerLiteral 
  OctalIntegerLiteral 
  HexIntegerLiteral

StrDecimalLiteral :::
  StrUnsignedDecimalLiteral 
  + StrUnsignedDecimalLiteral 
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :::
  Infinity 
  DecimalDigits . DecimalDigitsopt ExponentPartopt 
  DecimalDigits ExponentPartopt 
  DecimalDigits ExponentPartopt

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

ExponentPart ::
    ExponentIndicator SignedInteger

ExponentIndicator :: one of
    e E

SignedInteger ::
    DecimalDigits
    + DecimalDigits
    - DecimalDigits

HexIntegerLiteral ::
    0x HexDigits
    0X HexDigits

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

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 Classess

uri :::
    uriCharacters_opt

uriCharacters :::
    uriCharacter uriCharacters_opt

uriCharacter :::
    uriReserved
    uriUnescaped
    uriEscaped

uriReserved :: one of
    ; / ? : @ & = + $ ,

uriUnescaped :::
    uriAlpha
    DecimalDigit
uriMark

uriEscaped ::: % HexDigit HexDigit

uriAlpha ::: one of abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

uriMark ::: one of -.!~*'

A.8 Regular Expressions

Pattern[u] ::
  Disjunction[?u]

Disjunction[u] ::
  Alternative[?u]
  Alternative[?u] | Disjunction[?u]

Alternative[u] ::
  [empty]
  Alternative[?u] Term[?u]

Term[u] ::
  Assertion[?u]
  Atom[?u]
  Atom[?u] Quantifier

Assertion[u] ::
  ^
  $
  \ b
  \ B
  ( ? = Disjunction[?u] )
  ( ? ! Disjunction[?u] )

Quantifier ::
  QuantifierPrefix
  QuantifierPrefix ?
QuantifierPrefix ::
   *
   +
   ?
   { DecimalDigits }
   { DecimalDigits , }
   { DecimalDigits , DecimalDigits }

Atom[u] ::
   PatternCharacter

   .
   \ AtomEscape[?u]
   CharacterClass[?u]
   ( Disjunction[?u] )
   ( ? : Disjunction[?u] )

SyntaxCharacter :: one of
   ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
   SourceCharacter but not SyntaxCharacter

AtomEscape[u] ::
   DecimalEscape
   CharacterClassEscape
   CharacterEscape[?u]

CharacterEscape[u] ::
   ControlEscape
   c ControlLetter
   \ [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
RegExUnicodeEscapeSequence[\u] ::
  \[+\u\] u LeadSurrogate  \u TrailSurrogate
  \[+\u\] u LeadSurrogate
  \[+\u\] u TrailSurrogate
  \[+\u\] u NonSurrogate
  \[-\u\] u Hex4Digits
  \[+\u\] u{ HexDigits }

Each \u TrailSurrogate for which the choice of associated u LeadSurrogate is ambiguous shall be associated with the nearest possible u LeadSurrogate that would otherwise have no corresponding \u TrailSurrogate.

LeadSurrogate ::
  Hex4Digits but only if the SV of Hex4Digits is in the inclusive range 0xD800 to 0xDBFF

TrailSurrogate ::
  Hex4Digits but only if the SV of Hex4Digits is in the inclusive range 0xDC00 to 0xDFFF

NonSurrogate ::
  Hex4Digits but only if the SV of Hex4Digits is not in the inclusive range 0xD800 to 0xDFFF

IdentityEscape[\u] ::
  \[+\u\] SyntaxCharacter
  \[+\u\] /
  \[-\u\] SourceCharacter but not UnicodeIDContinue

DecimalEscape ::
  NonZeroDigit DecimalDigitsopt [lookahead \∉ DecimalDigit]

CharacterClassEscape :: one of
  d D s S w W

CharacterClass[\u] ::
  [ [lookahead \∉ \{ ^ \}] ClassRanges[\?\u] ]
  [ ^ ClassRanges[\?\u] ]

ClassRanges[\u] ::
  [empty]
  NonemptyClassRanges[\?\u]

NonemptyClassRanges[\u] ::
  ClassAtom[\?\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 based ECMAScript implementations. 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 11.8.3 is extended as follows except that this extension is not allowed for strict mode code:

**Syntax**

```
NumericLiteral ::
    DecimalLiteral
    BinaryIntegerLiteral
    OctalIntegerLiteral
    HexIntegerLiteral
    LegacyOctalIntegerLiteral

LegacyOctalIntegerLiteral ::
    0 OctalDigit
    LegacyOctalIntegerLiteral OctalDigit

DecimalIntegerLiteral ::
    0
    NonZeroDigit DecimalDigits<sub>opt</sub>
    NonOctalDecimalIntegerLiteral

NonOctalDecimalIntegerLiteral ::
    0 NonOctalDigit
    LegacyOctalLikeDecimalIntegerLiteral NonOctalDigit
    NonOctalDecimalIntegerLiteral DecimalDigit

LegacyOctalLikeDecimalIntegerLiteral ::
    0 OctalDigit
    LegacyOctalLikeDecimalIntegerLiteral OctalDigit

NonOctalDigit :: one of
    8 9
```
B.1.1.1 Static Semantics

- The MV of `LegacyOctalIntegerLiteral :: 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 :: 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 :: 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 11.8.4 is extended as follows except that this extension is not allowed for strict mode code:

Syntax

```
EscapeSequence ::
   CharacterEscapeSequence
   LegacyOctalEscapeSequence
   HexEscapeSequence
   UnicodeEscapeSequence

LegacyOctalEscapeSequence ::
   OctalDigit [lookahead != OctalDigit]
   ZeroToThree OctalDigit [lookahead != OctalDigit]
   FourToSeven OctalDigit
   ZeroToThree OctalDigit OctalDigit

ZeroToThree :: one of
   0 1 2 3

FourToSeven :: one of
   4 5 6 7
```
This definition of *EscapeSequence* is not used in strict mode or when parsing *TemplateCharacter*.

### B.1.2.1 Static Semantics

- The SV of *EscapeSequence :: LegacyOctalEscapeSequence* is the SV of the *LegacyOctalEscapeSequence*.
- The SV of *LegacyOctalEscapeSequence :: OctalDigit* is the code unit whose value is the MV of the *OctalDigit*.
- The SV of *LegacyOctalEscapeSequence :: ZeroToThree OctalDigit* is the code unit whose value is (8 times the MV of the *ZeroToThree*) plus the MV of the *OctalDigit*.
- The SV of *LegacyOctalEscapeSequence :: FourToSeven OctalDigit* is the code unit whose value is (8 times the MV of the *FourToSeven*) plus the MV of the *OctalDigit*.
- The SV of *LegacyOctalEscapeSequence :: ZeroToThree OctalDigit OctalDigit* is the code unit whose value is (64 (that is, $8^2$) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree :: 0* is 0.
- The MV of *ZeroToThree :: 1* is 1.
- The MV of *ZeroToThree :: 2* is 2.
- The MV of *ZeroToThree :: 3* is 3.
- The MV of *FourToSeven :: 4* is 4.
- The MV of *FourToSeven :: 5* is 5.
- The MV of *FourToSeven :: 6* is 6.
- The MV of *FourToSeven :: 7* is 7.

### B.1.3 HTML-like Comments

The syntax and semantics of 11.4 is extended as follows except that this extension is not allowed when parsing source code using the *goal symbol Module*:

**Syntax**

```latex
Comment ::
  MultiLineComment
  SingleLineComment
  SingleLineHTMLOpenComment
  SingleLineHTMLCloseComment
  SingleLineDelimitedComment

MultiLineComment ::
  /* FirstCommentLine_opt  LineTerminator MultiLineCommentChars_opt  */
  HTMLCloseComment_opt
```
FirstCommentLine ::
    SingleLineDelimitedCommentChars

SingleLineHTMLOpenComment ::
    <!— SingleLineCommentChars_opt

SingleLineHTMLCloseComment ::
    LineTerminatorSequence HTMLCloseComment

SingleLineDelimitedComment ::
    /* SingleLineDelimitedCommentChars_opt */

HTMLCloseComment ::
    WhiteSpaceSequence_opt SingleLineDelimitedCommentSequence_opt -->
        SingleLineCommentChars_opt

SingleLineDelimitedCommentChars ::
    SingleLineNotAsteriskChar SingleLineDelimitedCommentChars_opt
    * SingleLinePostAsteriskCommentChars_opt

SingleLineNotAsteriskChar ::
    SourceCharacter but not one of * or LineTerminator

SingleLinePostAsteriskCommentChars ::
    SingleLineNotForwardSlashOrAsteriskChar SingleLineDelimitedCommentChars_opt
    * SingleLinePostAsteriskCommentChars_opt

SingleLineNotForwardSlashOrAsteriskChar ::
    SourceCharacter but not one of / or * or LineTerminator

WhiteSpaceSequence ::
    WhiteSpace WhiteSpaceSequence_opt

SingleLineDelimitedCommentSequence ::
    SingleLineDelimitedComment WhiteSpaceSequence_opt
    SingleLineDelimitedCommentSequence_opt

Similar to a MultiLineComment that contains a line terminator code point, a SingleLineHTMLCloseComment is considered to be a LineTerminator for purposes of parsing by the syntactic grammar.

B.1.4 Regular Expressions Patterns
The syntax of 21.2.1 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**

\[
\begin{align*}
\text{Term}[U] & : \\
& \quad [+U] \text{Assertion}[+U] \\
& \quad [+U] \text{Atom}[+U] \\
& \quad [+U] \text{Atom}[+U] \quad \text{Quantifier} \\
& \quad [-U] \text{QuantifiableAssertion} \quad \text{Quantifier} \\
& \quad [-U] \text{Assertion}[-U] \\
& \quad [-U] \text{ExtendedAtom} \quad \text{Quantifier} \\
& \quad [-U] \text{ExtendedAtom} \\
\end{align*}
\]

\[
\begin{align*}
\text{Assertion}[U] & :: \\
& \quad ^
\end{align*}
\]

\[
\begin{align*}
& \quad $ \\
& \quad \backslash b \\
& \quad \backslash B \\
& \quad [+U] \left( \ ? = \text{Disjunction}[+U] \right) \\
& \quad [+U] \left( \ ? ! \text{Disjunction}[+U] \right) \\
& \quad [-U] \text{QuantifiableAssertion} \\
\end{align*}
\]

\[
\begin{align*}
\text{QuantifiableAssertion} & :: \\
& \quad \left( \ ? = \text{Disjunction}[-U] \right) \\
& \quad \left( \ ? ! \text{Disjunction}[-U] \right) \\
\end{align*}
\]

\[
\begin{align*}
\text{ExtendedAtom} & :: \\
& \quad . \\
& \quad \backslash \text{AtomEscape}[-U] \\
& \quad \text{CharacterClass}[-U] \\
& \quad \left( \text{Disjunction}[-U] \right) \\
& \quad \left( \ ? : \text{Disjunction}[-U] \right) \\
& \quad \text{InvalidBracedQuantifier} \\
& \quad \text{ExtendedPatternCharacter} \\
\end{align*}
\]
InvalidBracedQuantifier ::
   { DecimalDigits }
   { DecimalDigits , }
   { DecimalDigits , DecimalDigits }

ExtendedPatternCharacter ::
   SourceCharacter but not one of ^ $ . * + ? ( ) [ ]

AtomEscape[\u] ::
   [\u] DecimalEscape
   [~\u] DecimalEscape but only if the integer value of DecimalEscape is <= _NcapturingParens_
   CharacterClassEscape
   CharacterEscape[~\u]

CharacterEscape[\u] ::
   ControlEscape
   c ControlLetter
   \[lookahead \notin DecimalDigit\]
   HexEscapeSequence
   RegExpUnicodeEscapeSequence[?\u]
   [~\u] LegacyOctalEscapeSequence
   IdentityEscape[?\u]

IdentityEscape[\u] ::
   [\u] SyntaxCharacter
   [\u] /
   [~\u] SourceCharacter but not c

ClassEscape[\u] ::
   b
   [\u] -
   [~\u] c ClassControlLetter
   CharacterClassEscape
   CharacterEscape[?\u]

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 Pattern Semantics

The semantics of 21.2.2 is extended as follows:

Within 21.2.2.5 reference to “Atom :: ( Disjunction )” are to be interpreted as meaning “Atom :: ( Disjunction )” or “ExtendedAtom :: ( Disjunction )”.

Term (21.2.2.5) includes the following additional evaluation rules:

The production Term :: QuantifiableAssertion Quantifier evaluates the same as the production Term :: Atom Quantifier but with QuantifiableAssertion substituted for Atom.

The production Term :: ExtendedAtom Quantifier evaluates the same as the production Term :: Atom Quantifier but with ExtendedAtom substituted for Atom.

The production Term :: ExtendedAtom evaluates the same as the production Term :: Atom but with ExtendedAtom substituted for Atom.

Assertion (21.2.2.6) includes the following additional evaluation rule:

The production Assertion :: QuantifiableAssertion evaluates as follows:

1. Evaluate QuantifiableAssertion to obtain a Matcher m.
2. Return m.

Assertion (21.2.2.6) evaluation rules for the Assertion :: ( ? = Disjunction ) and Assertion :: ( ? ! Disjunction ) productions are also used for the QuantifiableAssertion productions, but with QuantifiableAssertion substituted for Assertion.

Atom (21.2.2.8) 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 are also added:

The production ExtendedAtom :: InvalidBracedQuantifier evaluates as follows:

1. Throw a SyntaxError exception.

The production ExtendedAtom :: ExtendedPatternCharacter evaluates as follows:

1. Let ch be the character represented by ExtendedPatternCharacter.
2. Let A be a one-element CharSet containing the character ch.
3. Call CharSetMatcher(A, false) and return its Matcher result.

CharacterEscape (21.2.2.10) includes the following additional evaluation rule:

The production CharacterEscape :: LegacyOctalEscapeSequence evaluates as follows:
1. Evaluate the SV of the *LegacyOctalEscapeSequence* (see B.1.2) to obtain a character \( ch \).
2. Return \( ch \).

**NonemptyClassRanges** (21.2.2.15) modifies the following evaluation rule:

The production **NonemptyClassRanges** :: **ClassAtom** - **ClassAtom** **ClassRanges** evaluates as follows:

1. Evaluate the first **ClassAtom** to obtain a CharSet \( A \).
2. Evaluate the second **ClassAtom** to obtain a CharSet \( B \).
3. Evaluate **ClassRanges** to obtain a CharSet \( C \).
4. Call **CharacterRangeOrUnion**\((A, B)\) and let \( D \) be the resulting CharSet.
5. Return the union of CharSets \( D \) and \( C \).

**NonemptyClassRangesNoDash** (21.2.2.16) modifies the following evaluation rule:

The production **NonemptyClassRangesNoDash** :: **ClassAtomNoDash** - **ClassAtom** **ClassRanges** evaluates as follows:

1. Evaluate **ClassAtomNoDash** to obtain a CharSet \( A \).
2. Evaluate **ClassAtom** to obtain a CharSet \( B \).
3. Evaluate **ClassRanges** to obtain a CharSet \( C \).
4. Call **CharacterRangeOrUnion**\((A, B)\) and let \( D \) be the resulting CharSet.
5. Return the union of CharSets \( D \) and \( C \).

**ClassEscape** (21.2.2.19) includes the following additional evaluation rule:

The production **ClassEscape** :: **c** **ClassControlLetter** evaluates as follows:

1. Let \( ch \) be the character matched by **ClassControlLetter**.
2. Let \( i \) be \( ch \)'s character value.
3. Let \( j \) be the remainder of dividing \( i \) by 32.
4. Let \( d \) be the character whose character value is \( j \).
5. Return the CharSet containing the single character \( d \).

### B.1.4.1.1 Runtime Semantics: **CharacterRangeOrUnion** \((A, B)\)

The abstract operation **CharacterRangeOrUnion** takes two CharSet parameters \( A \) and \( B \) and performs the following steps:

1. If **Unicode** is **false**, then
   a. If \( A \) does not contain exactly one character or \( B \) does not contain exactly one character, then
      i. Let \( C \) be the CharSet containing the single character - U+002D (HYPHEN-MINUS).
      ii. Return the union of CharSets \( A \), \( B \) and \( C \).
   2. Return **CharacterRange**\((A, B)\).
B.2 Additional Built-in Properties

When the ECMAScript host is a web browser the following additional properties of the standard built-in objects are defined.

B.2.1 Additional Properties of the Global Object

The entries in Table 67 are added to Table 7.

<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 (string)

The escape function is a property of the global object. It computes a new version of a String value in which certain code units have been replaced by a hexadecimal escape sequence.

For those code units being replaced whose value is 0x00FF or less, a two-digit escape sequence of the form %xx is used. For those characters being replaced whose code unit value is greater than 0x00FF, a four-digit escape sequence of the form %uxxxx is used.

The escape function is the %escape% intrinsic object. When the escape function is called with one argument string, the following steps are taken:

1. Set string to ?ToString(string).
2. Let length be the number of code units in string.
3. Let R be the empty string.
4. Let k be 0.
5. Repeat, while k < 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 a String containing the single code unit char.
   c. Else if char ≥ 256, then
      i. Let S be a String containing six code units "%uwxxyz" where wxyz are the code units of the four uppercase hexadecimal digits encoding char.
   d. Else char < 256,
Let $S$ be a String containing three code units "%xy" where $xy$ are the code units of two uppercase hexadecimal digits encoding char.

e. Set $R$ to a new String value computed by concatenating the previous value of $R$ and $S$.
f. Increase $k$ by 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 $k \neq$ length
   a. Let $c$ be the code unit at index $k$ within string.
   b. If $c$ is %, then
      i. If $k \leq$ length-6 and the code unit at index $k+1$ within string is u and the four code units at indices $k+2$, $k+3$, $k+4$, and $k+5$ within string are all hexadecimal digits, then
         1. Let $c$ be the code unit whose value is the integer represented by the four hexadecimal digits at indices $k+2$, $k+3$, $k+4$, and $k+5$ within string.
         2. Increase $k$ by 5.
      ii. Else if $k \leq$ length-3 and the two code units at indices $k+1$ and $k+2$ within string are both hexadecimal digits, then
          1. Let $c$ be the code unit whose value is the integer represented by two zeroes plus the two hexadecimal digits at indices $k+1$ and $k+2$ within string.
          2. Increase $k$ by 2.
   c. Set $R$ to a new String value computed by concatenating the previous value of $R$ and $c$.
   d. Increase $k$ by 1.
5. Return $R$.

B.2.2 Additional Properties of the Object.prototype Object
B.2.2.1 Object.prototype._proto_

Object.prototype._proto_ is an accessor property with attributes { [[Enumerable]]: false, [[Configurable]]: true }. The [[Get]] and [[Set]] attributes are defined as follows:

B.2.2.1.1 get Object.prototype._proto_

The value of the [[Get]] attribute is a built-in function that requires no arguments. It performs the following steps:

1. Let \( O \) be ? ToObject(\( \text{this} \) value).
2. Return ? \( O.[[GetPrototypeOf]]() \).

B.2.2.1.2 set Object.prototype._proto_

The value of the [[Set]] attribute is a built-in function that takes an argument proto. It performs the following steps:

1. Let \( O \) be ? RequireObjectCoercible(\( \text{this} \) value).
2. If Type(proto) is neither Object nor Null, return undefined.
3. If Type(\( O \)) is not Object, return undefined.
4. Let status be ? \( O.[[SetPrototypeOf]](\text{proto}) \).
5. If status is false, throw a TypeError exception.
6. Return undefined.

B.2.2.2 Object.prototype._defineGetter_ ( \( P, \text{getter} \) )

When the _defineGetter_ method is called with arguments \( P \) and getter, the following steps are taken:

1. Let \( O \) be ? ToObject(\( \text{this} \) value).
2. If IsCallable(getter) is false, throw a TypeError exception.
3. Let desc be PropertyDescriptor{[[Get]]: getter, [[Enumerable]]: true, [[Configurable]]: true}.
4. Let key be ? ToPropertyKey(\( P \)).
5. Perform ? DefinePropertyOrThrow(\( O, \text{key}, \text{desc} \)).
6. Return undefined.

B.2.2.3 Object.prototype._defineSetter_ ( \( P, \text{setter} \) )

When the _defineSetter_ method is called with arguments \( P \) and setter, the following steps are taken:

1. Let \( O \) be ? ToObject(\( \text{this} \) value).
2. If IsCallable(setter) is false, throw a TypeError exception.
3. Let desc be PropertyDescriptor{[[Set]]: setter, [[Enumerable]]: true, [[Configurable]]: true}. 
4. Let \textit{key} be \( \text{ToPropertyKey}(P) \).
5. Perform \( \text{DefinePropertyOrThrow}(O, \textit{key}, \text{desc}) \).
6. Return \textit{undefined}.

\textbf{B.2.2.4 Object.prototype.__lookupGetter__ (P)}

When the \texttt{__lookupGetter\_} method is called with argument \( P \), the following steps are taken:

1. Let \( O \) be \( \text{ToObject(this value)} \).
2. Let \( \textit{key} \) be \( \text{ToPropertyKey}(P) \).
3. Repeat,
   a. Let \( \textit{desc} \) be \( O.[\text{GetOwnProperty}][\textit{key}] \).
   b. If \( \textit{desc} \) is not \texttt{undefined}, then
      i. If \texttt{IsAccessorDescriptor(desc)} is \texttt{true}, return \( \textit{desc}[[\text{Get}]] \).
      ii. Return \texttt{undefined}.
   c. Set \( O \) to \( O.[\text{GetPrototypeOf}]() \).
   d. If \( O \) is \texttt{null}, return \texttt{undefined}.

\textbf{B.2.2.5 Object.prototype.__lookupSetter__ (P)}

When the \texttt{__lookupSetter\_} method is called with argument \( P \), the following steps are taken:

1. Let \( O \) be \( \text{ToObject(this value)} \).
2. Let \( \textit{key} \) be \( \text{ToPropertyKey}(P) \).
3. Repeat,
   a. Let \( \textit{desc} \) be \( O.[\text{GetOwnProperty}][\textit{key}] \).
   b. If \( \textit{desc} \) is not \texttt{undefined}, then
      i. If \texttt{IsAccessorDescriptor(desc)} is \texttt{true}, return \( \textit{desc}[[\text{Set}]] \).
      ii. Return \texttt{undefined}.
   c. Set \( O \) to \( O.[\text{GetPrototypeOf}]() \).
   d. If \( O \) is \texttt{null}, return \texttt{undefined}.

\textbf{B.2.3 Additional Properties of the String.prototype Object}

\textbf{B.2.3.1 String.prototype.substr (start, length)}

The \texttt{substr} method takes two arguments, \texttt{start} and \texttt{length}, and returns a substring of the result of converting the \texttt{this} object to a String, starting from index \texttt{start} and running for \texttt{length} code units (or through the end of the String if \texttt{length} is \texttt{undefined}). If \texttt{start} is negative, it is treated as \texttt{sourceLength+start} where \texttt{sourceLength} is the length of the String. The result is a String value, not a String object. The following steps are taken:
1. Let \( O \) be ? \text{RequireObjectCoercible}(\text{this value}).
2. Let \( S \) be ? \text{ToString}(O).
3. Let \( \text{intStart} \) be ? \text{ToInteger}(\text{start}).
4. If \( \text{length} \) is \text{undefined}, let \( \text{end} \) be \( +\infty \); otherwise let \( \text{end} \) be ? \text{ToInteger}(\text{length}).
5. Let \( \text{size} \) be the number of code units in \( S \).
6. If \( \text{intStart} < 0 \), let \( \text{intStart} \) be \( \text{max}(\text{size} + \text{intStart}, 0) \).
7. Let \( \text{resultLength} \) be \( \text{min}(\text{max}(\text{end}, 0), \text{size} - \text{intStart}) \).
8. If \( \text{resultLength} \leq 0 \), return the empty String "".
9. Return a String containing \( \text{resultLength} \) consecutive code units from \( S \) beginning with the code unit at index \( \text{intStart} \).

\textbf{NOTE} \quad \text{The \texttt{substr} 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.}

\textbf{B.2.3.2 String.prototype.\texttt{anchor} ( name )}

When the \texttt{anchor} method is called with argument \texttt{name}, the following steps are taken:

1. Let \( S \) be the \texttt{this} value.

\textbf{B.2.3.2.1 Runtime Semantics: \texttt{CreateHTML} ( string, tag, attribute, value )}

The abstract operation \texttt{CreateHTML} is called with arguments \texttt{string}, \texttt{tag}, \texttt{attribute}, and \texttt{value}. The arguments \texttt{tag} and \texttt{attribute} must be String values. The following steps are taken:

1. Let \( \text{str} \) be ? \text{RequireObjectCoercible}(\text{string}).
2. Let \( S \) be ? \text{ToString}(\text{str}).
3. Let \( p1 \) be the String value that is the concatenation of "<" and \texttt{tag}.
4. If \texttt{attribute} is not the empty String, then
   a. Let \( \text{V} \) be ? \text{ToString}(\text{value}).
   b. Let \texttt{escapedV} be the String value that is the same as \( \text{V} \) except that each occurrence of the code unit 0x0022 (QUOTATION MARK) in \( \text{V} \) has been replaced with the six code unit sequence "&quot;".
   c. Set \( p1 \) to the String value that is the concatenation of the following String values:
      - The String value of \( p1 \)
      - Code unit 0x0020 (SPACE)
      - The String value of \texttt{attribute}
      - Code unit 0x003D (EQUALS SIGN)
      - Code unit 0x0022 (QUOTATION MARK)
      - The String value of \texttt{escapedV}
      - Code unit 0x0022 (QUOTATION MARK)
5. Let \( p_2 \) be the String value that is the concatenation of \( p_1 \) and \\("\>\).  
6. Let \( p_3 \) be the String value that is the concatenation of \( p_2 \) and \( S \).  
7. Let \( p_4 \) be the String value that is the concatenation of \( p_3 \), \\("/<\>\), \( tag \), and \\("\>\).
8. Return \( p_4 \).

**B.2.3.3 String.prototype.big ()**

When the `big` method is called with no arguments, the following steps are taken:

1. Let \( S \) be the `this` value.  
2. Return `? CreateHTML(S, "big", "", "").`  

**B.2.3.4 String.prototype.blink ()**

When the `blink` method is called with no arguments, the following steps are taken:

1. Let \( S \) be the `this` value.  
2. Return `? CreateHTML(S, "blink", "", "").`  

**B.2.3.5 String.prototype.bold ()**

When the `bold` method is called with no arguments, the following steps are taken:

1. Let \( S \) be the `this` value.  
2. Return `? CreateHTML(S, "b", "", "").`  

**B.2.3.6 String.prototype.fixed ()**

When the `fixed` method is called with no arguments, the following steps are taken:

1. Let \( S \) be the `this` value.  
2. Return `? CreateHTML(S, "tt", "", "").`  

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

When the `fontcolor` method is called with argument `color`, the following steps are taken:

1. Let \( S \) be the `this` value.  

**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.

**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 \( \text{CreateHTML}(S, "sup", "", ")). \)

### B.2.4 Additional Properties of the Date.prototype Object

#### B.2.4.1 Date.prototype.getYear ( )

**NOTE** The \texttt{getFullYear} method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the \texttt{getYear} method is called with no arguments, the following steps are taken:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this} \text{ value}) \).
2. If \( t \) is \texttt{NaN}, return \texttt{NaN}.
3. Return \( \text{YearFromTime(ToLocalTime}(t)) - 1900 \).

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

**NOTE** The \texttt{getFullYear} method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the \texttt{setYear} method is called with one argument \textit{year}, the following steps are taken:

1. Let \( t \) be \( \text{thisTimeValue}(\text{this} \text{ value}) \).
2. If \( t \) is \texttt{NaN}, let \( t \) be \texttt{+0}; otherwise, let \( t \) be \text{LocalTime}(t).
3. Let \( y \) be \text{ToNumber}(\textit{year}).
4. If \( y \) is \texttt{NaN}, set the [[DateValue]] internal slot of this Date object to \texttt{NaN} and return \texttt{NaN}.
5. If \( 0 \leq \text{ToInteger}(y) \leq 99 \), let \textit{yyyy} be \text{ToInteger}(y) + 1900.
6. Else, let \textit{yyyy} be \textit{y}.
7. Let \( d \) be \text{MakeDay}(\textit{yyyy}, \text{MonthFromTime}(t), \text{DateFromTime}(t)).
8. Let \( \textit{date} \) be \text{UTC}(\text{MakeDate}(\textit{d}, \text{TimeWithinDay}(t)))
9. Set the [[DateValue]] internal slot of this Date object to \text{TimeClip}(\textit{date}).
10. Return the value of the [[DateValue]] internal slot of this Date object.

#### B.2.4.3 Date.prototype.toGMTString ( )

**NOTE** The property \texttt{toUTCString} is preferred. The \texttt{toGMTString} property is provided principally for compatibility with old code. It is recommended that the \texttt{toUTCString} property be used in new ECMAScript code.

The function object that is the initial value of \texttt{Date.prototype.toGMTString} is the same function object that is the initial value of \texttt{Date.prototype.toUTCString}.
B.2.5 Additional Properties of the RegExp.prototype Object

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

When the `compile` method is called with arguments `pattern` and `flags`, the following steps are taken:

1. Let `O` be the `this` value.
2. If `Type(O)` is not Object or `Type(O)` is Object and `O` does not have a `[[RegExpMatcher]]` internal slot, then
   a. Throw a `TypeError` exception.
3. If `Type(pattern)` is Object and `pattern` has a `[[RegExpMatcher]]` internal slot, then
   a. If `flags` is not `undefined`, throw a `TypeError` exception.
   b. Let `P` be `pattern`.([[OriginalSource]])
   c. Let `F` be `pattern`.([[OriginalFlags]])
4. Else,
   a. Let `P` be `pattern`.
   b. Let `F` be `flags`.

**NOTE** The `compile` method completely reinitializes the `this` object RegExp with a new pattern and flags. An implementation may interpret use of this method as an assertion that the resulting RegExp object will be used multiple times and hence is a candidate for extra optimization.

B.3 Other Additional Features

B.3.1 __proto__ Property Names in Object Initializers

The following Early Error rule is added to those in 12.2.6.1. When `ObjectLiteral` appears in a context where `ObjectAssignmentPattern` is required the Early Error rule is not applied. In addition, it is not applied when initially parsing a `CoverParenthesizedExpressionAndArrowParameterList` or a `CoverCallExpressionAndAsyncArrowHead`.

`ObjectLiteral : {  PropertyDefinitionList  }`
`ObjectLiteral : {  PropertyDefinitionList , }`

- It is a Syntax Error if `PropertyNameList` of `PropertyDefinitionList` contains any duplicate entries for `"__proto__"` and at least two of those entries were obtained from productions of the form
  `PropertyDefinition : PropertyName : AssignmentExpression`.

**NOTE** The List returned by `PropertyNameList` does not include string literal property names.
defined as using a `ComputedPropertyName`.

In 12.2.6.8 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. Let `exprValueRef` be the result of evaluating `AssignmentExpression`.
4. Let `propValue` be `? GetValue(exprValueRef)`.
5. If `propKey` is the String value `"__proto__"` and if `IsComputedPropertyKey(PropertyName)` is `false`, then
   a. If `Type(propValue)` is either `Object` or `Null`, then
      i. Return `object.[[SetPrototypeOf]](propValue)`.
   b. Return `NormalCompletion(empty)`. 
6. If `IsAnonymousFunctionDefinition(AssignmentExpression)` is `true`, then
   a. Let `hasNameProperty` be `? HasOwnProperty(propValue, "name")`.
   b. If `hasNameProperty` is `false`, perform `SetFunctionName(propValue, propKey)`.
7. Assert: `enumerable` is `true`.
8. 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, the grammar productions for `LabelledStatement` permits use of `FunctionDeclaration` as a `LabelledItem` but 13.13.1 includes an Early Error rule that produces a Syntax Error if that occurs. For web browser compatibility, that rule is modified with the addition of the 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.
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 9, 13, and 14 of this specification.
ECMAScript 2015 interoperability for the second and third use cases requires the following extensions to the clause 9, clause 14, clause 18.2.1 and clause 15.1.11 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 28:

1. 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 FunctionDeclaration f.
      ii. If replacing the FunctionDeclaration f with a VariableStatement that has F as a BindingIdentifier would not produce any Early Errors for func and F is not an element of parameterNames, then
          1. NOTE: A var binding for F is only instantiated here if it is neither a VarDeclaredName, the name of a formal parameter, or another FunctionDeclaration.
   2. If instantiatedVarNames does not contain F, then
      a. Perform ! varEnvRec.CreateMutableBinding(F, false).
      b. Perform varEnvRec.InitializeBinding(F, undefined).
      c. Append F to instantiatedVarNames.
   3. When the FunctionDeclaration f is evaluated, perform the following steps in place of the FunctionDeclaration Evaluation algorithm provided in 14.1.21:
      a. Let fenv be the running execution context’s VariableEnvironment.
      b. Let fenvRec be fenv’s EnvironmentRecord.
      c. Let benv be the running execution context’s LexicalEnvironment.
      d. Let benvRec be benv’s EnvironmentRecord.
      e. Let fobj be ! benvRec.GetBindingValue(F, false).
      f. Perform ! fenvRec.SetMutableBinding(F, fobj, false).
      g. Return NormalCompletion(empty).

### B.3.3.2 Changes to GlobalDeclarationInstantiation

During GlobalDeclarationInstantiation the following steps are performed in place of step 14:

1. Let strict be IsStrict of script.
2. If strict is false, then
a. Let \textit{declFunctionOrVarNames} be a new empty \textit{List}.
b. Append to \textit{declFunctionOrVarNames} the elements of \textit{declFunctionNames}.
c. Append to \textit{declFunctionOrVarNames} the elements of \textit{declVarNames}.
d. For each \textit{FunctionDeclaration} \textit{f} that is directly contained in the \textit{StatementList} of a \textit{Block}, \textit{CaseClause}, or \textit{DefaultClause} Contained within \textit{script}, do
   i. Let \textit{F} be StringValue of the \textit{BindingIdentifier} of \textit{FunctionDeclaration} \textit{f}.
   ii. If replacing the \textit{FunctionDeclaration} \textit{f} with a \textit{VariableStatement} that has \textit{F} as a 
       \textit{BindingIdentifier} would not produce any Early Errors for \textit{script}, then
       1. If \textit{envRec}.HasLexicalDeclaration(\textit{F}) is \textit{false}, then
          a. Let \textit{fnDefinable} be ? \textit{envRec}.CanDeclareGlobalFunction(\textit{F}).
          b. If \textit{fnDefinable} is \textit{true}, then
             i. NOTE: A var binding for \textit{F} is only instantiated here if it is neither a 
                VarDeclaredName nor the name of another \textit{FunctionDeclaration}.
             ii. If \textit{declFunctionOrVarNames} does not contain \textit{F}, then
                i. Perform ? \textit{envRec}.CreateGlobalFunctionBinding(\textit{F}, \textit{undefined, false}).
                ii. Append \textit{F} to \textit{declFunctionOrVarNames}.
             iii. When the \textit{FunctionDeclaration} \textit{f} is evaluated, perform the following 
                  steps in place of the \textit{FunctionDeclaration} Evaluation algorithm provided in 14.1.21:
                 i. Let \textit{genv} be the \textit{running execution context}'s VariableEnvironment.
                 ii. Let \textit{genvRec} be \textit{genv}'s EnvironmentRecord.
                 iii. Let \textit{benv} be the \textit{running execution context}'s LexicalEnvironment.
                 iv. Let \textit{benvRec} be \textit{benv}'s EnvironmentRecord.
                 v. Let \textit{fobj} be ! \textit{benvRec}.GetBindingValue(\textit{F}, \textit{false}).
                 vii. Return \textit{NormalCompletion} (empty).

\section*{B.3.3.3 Changes to EvalDeclarationInstantiation}

During \textit{EvalDeclarationInstantiation} the following steps are performed in place of step 9:

1. If \textit{strict} is \textit{false}, then
   a. Let \textit{declFunctionOrVarNames} be a new empty \textit{List}.
   b. Append to \textit{declFunctionOrVarNames} the elements of \textit{declFunctionNames}.
   c. Append to \textit{declFunctionOrVarNames} the elements of \textit{declVarNames}.
   d. For each \textit{FunctionDeclaration} \textit{f} that is directly contained in the \textit{StatementList} of a \textit{Block}, \textit{CaseClause}, or \textit{DefaultClause} Contained within \textit{body}, do
      i. Let \textit{F} be StringValue of the \textit{BindingIdentifier} of \textit{FunctionDeclaration} \textit{f}.
      ii. If replacing the \textit{FunctionDeclaration} \textit{f} with a \textit{VariableStatement} that has \textit{F} as a 
          \textit{BindingIdentifier} would not produce any Early Errors for \textit{body}, then
1. Let $bindingExists$ be false.
2. Let $thisLex$ be $lexEnv$.
3. Assert: The following loop will terminate.
4. Repeat, while $thisLex$ is not the same as $varEnv$,
   a. Let $thisEnvRec$ be $thisLex$'s EnvironmentRecord.
   b. If $thisEnvRec$ is not an object Environment Record, then
      i. If $thisEnvRec$.HasBinding($F$) is true, then
      i. Let $bindingExists$ be true.
   c. Let $thisLex$ be $thisLex$'s outer environment reference.
5. If $bindingExists$ is false and $varEnvRec$ is a global Environment Record, then
   a. If $varEnvRec$.HasLexicalDeclaration($F$) is false, then
      i. Let $fnDefinable$ be ? $varEnvRec$.CanDeclareGlobalFunction($F$).
   b. Else,
      i. Let $fnDefinable$ be false.
6. Else,
   a. Let $fnDefinable$ be true.
7. If $bindingExists$ is false and $fnDefinable$ is true, then
   a. If $declaredFunctionOrVarNames$ does not contain $F$, then
      i. If $varEnvRec$ is a global Environment Record, then
         i. Perform ? $varEnvRec$.CreateGlobalFunctionBinding($F$, undefined, true).
      ii. Else,
         i. Let $bindingExists$ be $varEnvRec$.HasBinding($F$).
      ii. If $bindingExists$ is false, then
         i. Perform ! $varEnvRec$.CreateMutableBinding($F$, true).
         ii. Perform ! $varEnvRec$.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 14.1.21:
      i. Let $genv$ be the running execution context's VariableEnvironment.
      ii. Let $genvRec$ be $genv$'s EnvironmentRecord.
      iii. Let $benv$ be the running execution context's LexicalEnvironment.
      iv. Let $benvRec$ be $benv$'s EnvironmentRecord.
      v. Let $fobj$ be ! $benvRec$.GetBindingValue($F$, false).
      vii. Return NormalCompletion(empty).

B.3.3.4  Changes to Block Static Semantics: Early Errors

For web browser compatibility, that rule is modified with the addition of the highlighted text:
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.

### B.3.3.5 Changes to switch Statement Static Semantics: Early Errors

For web browser compatibility, that rule is modified with the addition of the highlighted text:

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

- 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.

### B.3.3.6 Changes to BlockDeclarationInstantiation

During BlockDeclarationInstantiation the following steps are performed in place of step 4.b.iii:

1. If \(\text{envRec.HasBinding(fn)}\) is true, then
   a. Assert: \(d\) is a FunctionDeclaration.
   b. Perform \(\text{envRec.SetMutableBinding(fn, fo, false)}\).
2. Else,
   a. Perform \(\text{envRec.InitializeBinding(fn, fo)}\).

### B.3.4 FunctionDeclarations in IfStatement Statement Clauses

The following augments the IfStatement production in 13.6:

\[
\text{IfStatement}[\text{Yield, Await, Return}] : \begin{align*}
\text{if ( Expression} & [\text{+In, ?Yield, ?Await}] \text{ ) } \\
\text{FunctionDeclaration} & [\text{?Yield, ?Await, ~Default}] \text{ else } \\
\text{Statement} & [\text{?Yield, ?Await, ?Return}] \text{ else } \\
\text{if ( Expression} & [\text{+In, ?Yield, ?Await}] \text{ ) } \\
\text{FunctionDeclaration} & [\text{?Yield, ?Await, ~Default}] \text{ else } \\
\text{FunctionDeclaration} & [\text{?Yield, ?Await, ~Default}] \text{ else } \\
\text{if ( Expression} & [\text{+In, ?Yield, ?Await}] \text{ ) } \\
\text{FunctionDeclaration} & [\text{?Yield, ?Await, ~Default}] \text{ else } \\
\text{FunctionDeclaration} & [\text{?Yield, ?Await, ~Default}] \text{ else } \\
\end{align*}
\]
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 `StatementList itemList` of a `BlockStatement` occupying that position in the source code. The semantics of such a synthetic `BlockStatement` includes the web legacy compatibility semantics specified in B.3.3.

### B.3.5 VariableStatements in Catch Blocks

The content of subclause 13.15.1 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` and that element is only bound by a `VariableStatement`, the `VariableDeclarationList` of a for statement, the `ForBinding` of a for-in statement, or the `BindingIdentifier` of a for-in statement.

**NOTE** The `Block` of a `Catch` clause may contain `var` declarations that bind a name that is also bound by the `CatchParameter`. At runtime, such bindings are instantiated in the VariableDeclarationEnvironment. They do not shadow the same-named bindings introduced by the `CatchParameter` and hence the `Initializer` for such `var` declarations will assign to the corresponding catch parameter rather than the `var` binding. The relaxation of the normal static semantic rule does not apply to names only bound by for-of statements.

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 18.2.1.3 as follows:

Step 5.d.ii.2.a.i is replaced by:

1. If `thisEnvRec` is not the `Environment Record` for a `Catch` clause, throw a `SyntaxError` exception.
2. If `name` is bound by any syntactic form other than a `FunctionDeclaration`, a `VariableStatement`, the `VariableDeclarationList` of a for statement, the `ForBinding` of a for-in statement, or the `BindingIdentifier` of a for-in statement, throw a `SyntaxError` exception.

Step 9.d.ii.4.b.i.i is replaced by:

1. If `thisEnvRec` 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 `IterationStatement` production in 13.7:

```javascript
IterationStatement[Yield, Await, Return] :
```

This production only applies when parsing `non-strict code`.

The static semantics of `ContainsDuplicateLabels` in 13.7.5.3 are augmented with the following:

```javascript
IterationStatement : for ( var BindingIdentifier Initializer in Expression ) Statement
```

1. Return `ContainsDuplicateLabels` of `Statement` with argument `labelSet`.

The static semantics of `ContainsUndefinedBreakTarget` in 13.7.5.4 are augmented with the following:

```javascript
IterationStatement : for ( var BindingIdentifier Initializer in Expression ) Statement
```

1. Return `ContainsUndefinedBreakTarget` of `Statement` with argument `labelSet`.

The static semantics of `ContainsUndefinedContinueTarget` in 13.7.5.5 are augmented with the following:

```javascript
IterationStatement : for ( var BindingIdentifier Initializer in Expression ) Statement
```

1. Return `ContainsUndefinedContinueTarget` of `Statement` with arguments `iterationSet` and « ».

The static semantics of `IsDestructuring` in 13.7.5.6 are augmented with the following:

```javascript
BindingIdentifier :
    Identifier
    yield
    await
```

1. Return `false`.

The static semantics of `VarDeclaredNames` in 13.7.5.7 are augmented with the following:

```javascript
IterationStatement : 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 13.7.5.8 are augmented with the following:

```javascript
IterationStatement : for ( var BindingIdentifier Initializer in Expression ) Statement
```
1. Let \textit{declarations} be a \texttt{List} containing \texttt{BindingIdentifier}.
2. Append to \textit{declarations} the elements of the VarScopedDeclarations of \texttt{Statement}.
3. Return \textit{declarations}.

The runtime semantics of LabelledEvaluation in 13.7.5.11 are augmented with the following:

\textsl{IterationStatement} : \texttt{for ( var BindingIdentifier Initializer in Expression ) Statement}

1. Let \textit{bindingId} be \texttt{StringValue} of \texttt{BindingIdentifier}.
2. Let \textit{lhs} be \texttt{? ResolveBinding(bindingId)}.
3. Let \textit{rhs} be the result of evaluating \texttt{Initializer}.
4. Let \textit{value} be \texttt{? GetValue(rhs)}.
5. If \texttt{IsAnonymousFunctionDefinition(Initializer)} is \texttt{true}, then
   a. Let \textit{hasNameProperty} be \texttt{? HasOwnProperty(value, "name")}.
   b. If \textit{hasNameProperty} is \texttt{false}, perform \texttt{SetFunctionName(value, bindingId)}.
6. Perform \texttt{? PutValue(lhs, value)}.
7. Let \textit{keyResult} be \texttt{? ForIn/OfHeadEvaluation(« », Expression, enumerate)}.
8. Return \texttt{? ForIn/OfBodyEvaluation(BindingIdentifier, Statement, keyResult, enumerate, varBinding, labelSet)}.

C The Strict Mode of ECMAScript

The strict mode restriction and exceptions

- \texttt{implements, interface, let, package, private, protected, public, static, and yield} are reserved words within \texttt{strict mode code}. (11.6.2).
- A conforming implementation, when processing \texttt{strict mode code}, must not extend, as described in B.1.1, the syntax of \texttt{NumericLiteral} to include \texttt{LegacyOctalIntegerLiteral}, nor extend the syntax of \texttt{DecimalIntegerLiteral} to include \texttt{NonOctalDecimalIntegerLiteral}.
- A conforming implementation, when processing \texttt{strict mode code}, may not extend the syntax of \texttt{EscapeSequence} to include \texttt{LegacyOctalEscapeSequence} as described in B.1.2.
- Assignment to an undeclared identifier or otherwise unresolvable reference does not create a property in the \texttt{global object}. When a simple assignment occurs within \texttt{strict mode code}, its \texttt{LeftHandSideExpression} must not evaluate to an unresolvable \texttt{Reference}. If it does a \texttt{ReferenceError} exception is thrown (6.2.4.2). The \texttt{LeftHandSideExpression} also may not be a reference to a data property with the attribute value \{[[Writable]]: \texttt{false}\}, to an accessor property with the attribute value \{[[Set]]: \texttt{undefined}\}, nor to a non-existent property of an object whose [[Extensible]] internal slot has the value \texttt{false}. In these cases a \texttt{TypeError} exception is thrown (12.15).
- The identifier \texttt{eval} or \texttt{arguments} may not appear as the \texttt{LeftHandSideExpression} of an Assignment operator (12.15) or of a \texttt{UpdateExpression} (12.4) or as the \texttt{UnaryExpression} operated upon by a Prefix
Arguments objects for strict functions define a non-configurable accessor property "calle" which throws a **TypeError** exception on access (9.4.4.6).

Arguments objects for strict functions do not dynamically share their array indexed property values with the corresponding formal parameter bindings of their functions. (9.4.4).

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. (9.2.12).

- It is a **SyntaxError** if the **IdentifierName** **eval** or the **IdentifierName** **arguments** occurs as a **BindingIdentifier** within strict mode code (12.1.1).
- Strict mode eval code cannot instantiate variables or functions in the variable environment of the caller to eval. Instead, a new variable environment is created and that environment is used for declaration binding instantiation for the eval code (18.2.1).
- If **this** is evaluated within strict mode code, then the **this** value is not coerced to an object. A **this** value of **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 (9.2.1.2, 19.2.3.1, 19.2.3.3).
- 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 (12.5.3.1).
- When a **delete** operator occurs within strict mode code, a **TypeError** is thrown if the property to be deleted has the attribute {[[Configurable]]: **false**} (12.5.3.2).
- Strict mode code may not include a **WithStatement**. The occurrence of a **WithStatement** in such a context is a **SyntaxError** (13.11.1).
- It is a **SyntaxError** if a **TryStatement with a Catch** occurs within strict mode code and the **Identifier** of the **Catch** is **eval** or **arguments** (13.15.1).
- 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** (14.1.2, 19.2.1.1.1).
- 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 Corrections and Clarifications in ECMAScript 2015 with Possible Compatibility Impact

8.1.1.4.15-8.1.1.4.18 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.

9.4.2.1: 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.

20.3.1.15: Previous editions permitted the TimeClip abstract operation to return either +0 or -0 as the representation of a 0 time value. ECMAScript 2015 specifies that +0 always returned. This means that for ECMAScript 2015 the time value of a Date object is never observably -0 and methods that return time values never return -0.

20.3.1.16: If a time zone offset is not present, the local time zone is used. Edition 5.1 incorrectly stated that a missing time zone should be interpreted as "z".

20.3.4.36: If the year cannot be represented using the Date Time String Format specified in 20.3.1.16 a RangeError exception is thrown. Previous editions did not specify the behaviour for that case.

20.3.4.41: Previous editions did not specify the value returned by Date.prototype.toString when this time value is NaN. ECMAScript 2015 specifies the result to be the String value is "Invalid Date".

21.2.3.1, 21.2.3.2.4: 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 "/".

21.2.5.6, 21.2.5.8: 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 is flag 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.

22.1.3.25, 22.1.3.25.1: 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 +0 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-dependent. In practice, implementations call ToNumber.

E Additions and Changes That Introduce Incompatibilities with Prior Editions
7.1.3.1: 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`.

6.2.4: In ECMAScript 2015, Function calls are not allowed to return a `Reference` value.

11.6: 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.

11.9.1: 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.

12.2.6.1: In ECMAScript 2015, it is no longer an `early error` to have duplicate property names in Object Initializers.

12.15.1: 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.

13.2: 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.

13.5: 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`.

13.6.7: 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`.

13.7: 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`.

13.7: 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`.

13.7: 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.

13.7: 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`.

13.11.7: In ECMAScript 2015, the normal completion value of a `WithStatement` is never the value `empty`. If evaluation of the `Statement` part of a `WithStatement` produces a normal completion whose value is `empty`, the completion value of the `WithStatement` is `undefined`.

13.12.11: 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`.

13.15: 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.

13.15, 18.2.1.3: 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.

13.15.8: 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.

14.3.8 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.

19.1.2.6: 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.

19.1.2.7: 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.
19.1.2.9: 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.

19.1.2.11: In ECMAScript 2015, if the argument to `Object.getPrototypeOf` is not an object an attempt is made to coerce the argument using `ToObject`. If the coercion is successful the result is used in place of the original argument value. In the previous edition, a non-object argument always causes a `TypeError` to be thrown.

19.1.2.13: 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.

19.1.2.14: 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.

19.1.2.15: 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.

19.1.2.16: 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.

19.1.2.17: 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.

19.1.2.19: 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.

19.2.3.2: 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 `%FunctionPrototype%`.

19.2.4.1: In ECMAScript 2015, the `length` property of function instances is configurable. In previous editions it was non-configurable.

19.5.6.2: In ECMAScript 2015, the `[[Prototype]]` internal slot of a `NativeError` constructor is the `Error` constructor. In previous editions it was the `Function` prototype object.
20.3.4 In ECMAScript 2015, the Date prototype object is not a Date instance. In previous editions it was a Date instance whose TimeValue was NaN.

21.1.3.10 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.

21.1.3.24 and 21.1.3.26 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.

21.1.3.27 In ECMAScript 2015, the `String.prototype.trim` method is defined to recognize white space code points that may exist outside of the Unicode BMP. However, as of Unicode 7 no such code points are defined. In previous editions such code points would not have been recognized as white space.

21.2.3.1 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`.

21.2.5 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.

21.2.5 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.

F Bibliography

7. IANA Time Zone Database, available at <https://www.iana.org/time-zones>
8. ISO 8601:2004(E) Data elements and interchange formats – Information interchange — Representation of dates and times

G Copyright & Software License

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

Copyright Notice

© 2017 Ecma International

This draft document may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as needed for the purpose of developing any document or deliverable produced by Ecma International.

This disclaimer is valid only prior to final version of this document. After approval all rights on the standard are reserved by Ecma International.

The limited permissions are granted through the standardization phase and will not be revoked by Ecma International or its successors or assigns during this time.
This document and the information contained herein is provided on an "AS IS" basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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 http://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.