Draft

Ecma/TC39/2013/0xx

Standard ECMA-262

6th Edition / Draft July 15, 2013

Draft

ECMAScript Language Specification

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

Product: Draft for 6th Edition
Component: choose an appropriate one
Version: Rev 16, July 15, 2013 Draft

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Contents

Page

DRAFT

Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.

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

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

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

Since publication of the third edition, ECMAScript has achieved massive adoption in conjunction with the World Wide Web where it has become the programming language that is supported by essentially all web browsers. Significant work was done to develop a fourth edition of ECMAScript. Although that work was not completed and not published[1] as the fourth edition of ECMAScript, it informs continuing evolution of the language. The fifth edition of ECMAScript (published as ECMA-262 5th edition) codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the third edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security.

The edition 5.1 of the ECMAScript Standard has been fully aligned with the third edition of the international standard ISO/IEC 16262:2011.

This present sixth edition of the Standard………

ECMAScript is a vibrant language and the evolution of the language is not complete. Significant technical enhancement will continue with future editions of this specification.

This Ecma Standard has been adopted by the General Assembly of <month> <year>.

---

[1] Note: Please note that for ECMAScript Edition 4 the Ecma standard number "ECMA-262 Edition 4" was reserved but not used in the Ecma publication process. Therefore "ECMA-262 Edition 4" as an Ecma International publication does not exist.

ECMAScript Language Specification

1 Scope

This Standard defines the ECMAScript scripting language.

2 Conformance

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax and semantics described in this specification.

A conforming implementation of this Standard shall interpret characters in conformance with the Unicode Standard, Version 5.1.0 or later and ISO/IEC 10646. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the Unicode set, collection 10646.

Deleted: 3

Deleted: -1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3

Deleted: BMP subset

Deleted: 300

Deleted: If the adopted encoding form is no otherwise specified, it presumed to be the UT 16 encoding form.

A conforming implementation of ECMAScript is permitted to provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript is permitted to 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 is permitted to support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the "future reserved words" listed in 7.6.1.2 of this specification.

3 Normative references

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

ISO/IEC 9899:1996, *Programming Languages – C, including amendment 1 and technical corrigenda 1 and 2*

ISO/IEC 10646:2003: *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005, Amendment 2:2006, Amendment 3:2008, and Amendment 4:2008*, plus additional amendments and corrigenda, or successor

The Unicode Standard, Version 5.0, as amended by Unicode 5.1.0, or successor

Unicode Standard Annex #15, Unicode Normalization Forms, version Unicode 5.1.0, or successor

Unicode Standard Annex #31, Unicode Identifiers and Pattern Syntax, version Unicode 5.1.0, or successor

Deleted: ISO/IEC 10646-1:1993, *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus its amendments ar corrigenda*

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.

Deleted: *host*

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

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

Deleted: can

Deleted: ,

Deleted: and t

Deleted: scripting

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

Gosling, James, Bill Joy and Guy Steele. The Java™ Language Specification. Addison Wesley Publishing Co., 1996.

Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227–241, Orlando, FL, October 1987.

IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

4.1 Web Scripting

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

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

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

4.2 Language 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. An ECMAScript ***object*** is a collection of ***properties*** each with zero or more ***attributes*** that determine how each property can be used—for example, when the Writable attribute for a property is set to **false**, any attempt by executed ECMAScript code to change the value of the property fails. Properties are containers that hold other objects, ***primitive values***, or ***functions***. A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, and **String**; an object is a member of the remaining built-in type **Object**; and a function is a callable object. A function that is associated with an object via a property is a ***method***.

ECMAScript defines a collection of ***built-in objects*** that round out the definition of ECMAScript entities. These built-in objects include the global object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, the **Date** object, the **RegExp** object, the **JSON** object, and the Error objects **Error, EvalError**, **RangeError, ReferenceError, SyntaxError, TypeError** and **URIError**.

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

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

4.2.1 Objects

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

Comment [AWB101]: This description probably need to be tweaked in light of new features such as class declarations and expli exposure of the [[Prototype]] property

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.

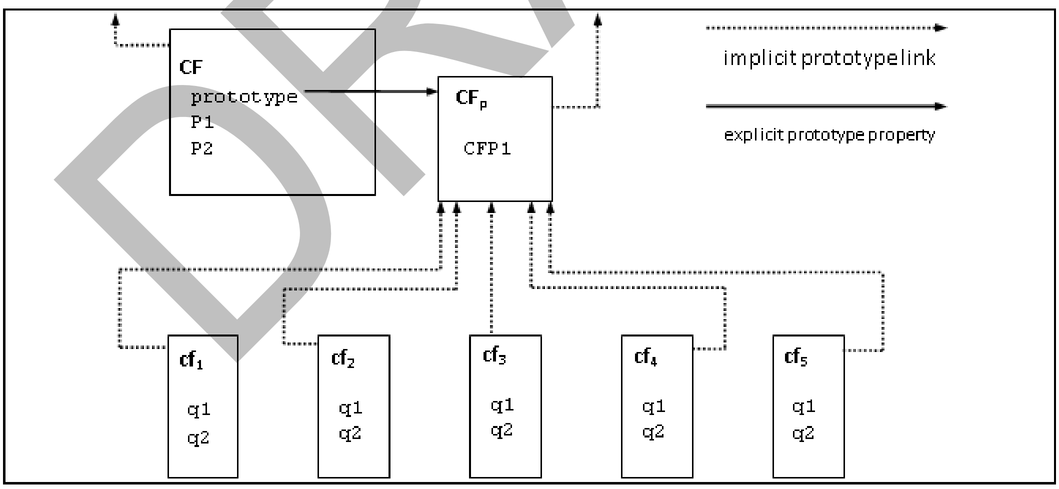

Figure 1 — Object/Prototype Relationships

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.

Deleted: and

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

Deleted: Figure 1

CF is a constructor (and also an object). Five objects have been created by using `new` expressions: **cf**$_1$, **cf**$_2$, **cf**$_3$, **cf**$_4$, and **cf**$_5$. Each of these objects contains properties named `q1` and `q2`. The dashed lines represent the implicit prototype relationship; so, for example, **cf**$_3$'s prototype is **CF**$_p$. The constructor, **CF**, has two properties itself, named `P1` and `P2`, which are not visible to **CF**$_p$, **cf**$_1$, **cf**$_2$, **cf**$_3$, **cf**$_4$, or **cf**$_5$. The property named `CFP1` in **CF**$_p$ is shared by **cf**$_1$, **cf**$_2$, **cf**$_3$, **cf**$_4$, and **cf**$_5$ (but not by **CF**), as are any properties found in **CF**$_p$'s implicit prototype chain that are not named `q1`, `q2`, or `CFP1`. Notice that there is no implicit prototype link between **CF** and **CF**$_p$.

Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for **cf**$_1$, **cf**$_2$, **cf**$_3$, **cf**$_4$, and **cf**$_5$ by assigning a new value to the property in **CF**$_p$.

4.2.2 The Strict Variant of ECMAScript

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

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

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

4.3 Terms and definitions

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

4.3.1
type
set of data values as defined in Clause 8 of this specification

4.3.2
primitive value
member of one of the types Undefined, Null, Boolean, Number, or String as defined in Clause 8

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

4.3.3
object
member of the type Object

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

4.3.4
constructor
function object that creates and initialises objects

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

4.3.5
prototype
object that provides shared properties for other objects

NOTE When a constructor creates an object, that object implicitly references the constructor's "**`prototype`**" property for the purpose of resolving property references. The constructor's "**`prototype`**" property can be referenced by the program expression *`constructor`*`.`**`prototype`**, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype. Alternatively, a new object may be created with an explicitly specified prototype by using the **`Object.create`** built-in function.

4.3.6
ordinary object
object that has the default behaviour for the internal methods that must be supported by all ECMAScript objects.

4.3.7
exotic object
object that has some alternative behaviour for one or more of the internal methods that must be supported by all ECMAScript objects.

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

4.3.8
standard object
object whose semantics are defined by this specification.

4.3.9
built-in object
object supplied by an ECMAScript implementation, independent of the host environment, that is present at the start of the execution of an ECMAScript program

NOTE Standard built-in objects are defined in this specification, and an ECMAScript implementation may specify and define others. A *built-in constructor* is a built-in object that is also a constructor.

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

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

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

4.3.13
Null type
type whose sole value is the null value

4.3.14
Boolean value
member of the Boolean type

Deleted: native

Deleted: in an ECMAScript implementation whose semantics are fully defined by this specification rather than by the host environment

Deleted: NOTE Standard native objects are defined in this specification. Some native objects are built-in; others may be constructe during the course of execution of an ECMAScript program.¶

Deleted: 7

Deleted: Every built-in object is a native obj

Deleted: 4.3.8¶
host exotic object¶
object that has some alternative behaviour fo one or more of the internal methods that mus be supported by all ECMAScript objects.supplied by the host environment to complete the execution environment of ECMAScript¶
NOTE Any object that is not an native ordina object is an host exotic object.¶

Deleted: 9

Deleted: 10

Deleted: 11

Deleted: 12

Deleted: 13

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

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

Deleted: 14

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

Deleted: 15

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 data property 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

Deleted: 16

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

Deleted: 17

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

Deleted: 18

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 data property whose value is the String value. A String object can be coerced to a String value by calling the `String` constructor as a function (15.5.1).

4.3.20
Number value
primitive value corresponding to a double-precision 64-bit binary format IEEE 754 value

Deleted: 19

NOTE A Number value is a member of the Number type and is a direct representation of a number.

4.3.21
Number type
set of all possible Number values including the special “Not-a-Number” (NaN) value, positive infinity, and negative infinity

Deleted: 20

Deleted: s

4.3.22
Number object
member of the Object type that is an instance of the standard built-in `Number` constructor

Deleted: 21

NOTE A Number object is created by using the `Number` constructor in a `new` expression, supplying a Number value as an argument. The resulting object has an internal data property whose value is the Number value. A Number object can be coerced to a Number value by calling the `Number` constructor as a function (15.7.1).

4.3.23
Infinity
number value that is the positive infinite Number value

Deleted: 22

4.3.24
NaN
number value that is a IEEE 754 “Not-a-Number” value

Deleted: 23

4.3.25
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.

4.3.26
built-in function
built-in object that is a function

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

4.3.27
property
association between a name and a value that is a part of an object

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.

4.3.28
method
function that is the value of a property

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

4.3.29
built-in method
method that is a built-in function

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

4.3.30
attribute
internal value that defines some characteristic of a property

4.3.31
own property
property that is directly contained by its object

4.3.32
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

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.

Deleted: 24
Deleted: is an instance of the standard built-**`Function`** constructor and that
Deleted: named
Deleted: 25
Deleted: 26
Deleted: 27
Deleted: 28
Deleted: 29
Deleted: 30
Deleted: 31
Deleted:
Deleted: one

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 7. This grammar has as its terminal symbols characters (Unicode code units) that conform to the rules for *SourceCharacter* defined in Clause 6. It defines a set of productions, starting from the goal symbol *InputElementDiv* or *InputElementRegExp*, that describe how sequences of such characters are translated into a sequence of input elements.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion (7.9). 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 15.10. This grammar also has as its terminal symbols the characters as defined by *SourceCharacter*. It defines a set of productions, starting from the goal symbol *Pattern*, that describe how sequences of characters are translated into regular expression patterns.

Productions of the lexical and RegExp grammars are distinguished by having two colons "**::**" as separating punctuation. The lexical and RegExp grammars share some productions.

5.1.3 The Numeric String Grammar

Another grammar is used for translating Strings into numeric values. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols *SourceCharacter*. This grammar appears in 9.3.1.

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

5.1.4 The Syntactic Grammar

The *syntactic grammar* for ECMAScript is given in clauses 11, 12, 13 and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (5.1.2). It defines a set of productions, starting from the goal symbol *Script*, that describe how sequences of tokens can form syntactically correct independent components of an ECMAScript programs.

Deleted: *Program*

When a stream of characters is to be parsed as an ECMAScript script, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntactic grammar. The script is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *Script*, with no tokens left over.

Deleted: program

Deleted: program

Deleted: *Program*

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

The syntactic grammar as presented in clauses 11, 12, 13 and 14 is actually not a complete account of which token sequences are accepted as correct ECMAScript scripts. Certain additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if a terminator character appears in certain "awkward" places.

Deleted: programs

In certain cases in order to avoid ambiguities the syntactic grammar uses generalize productions that permit token sequences that are not valid ECMAScript scripts. For example, this technique is used in with object literals and object destructuring patterns. In such cases a more restrictive *supplemental grammar* is provided that further restricts the acceptable token sequences. In certain contexts, when explicitly specific, the input elements corresponding to such a production are parsed again using a goal symbol of a supplemental grammar. The script is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the supplemental goal symbol, with no tokens left over.

Deleted: program

Deleted: s

Deleted:

Deleted: such situations

Deleted: is

Deleted: program

5.1.5 The JSON Grammar

The JSON grammar is used to translate a String describing a set of ECMAScript objects into actual objects. The JSON grammar is given in 15.12.1.

The JSON grammar consists of the JSON lexical grammar and the JSON syntactic grammar. The JSON lexical grammar is used to translate character sequences into tokens and is similar to parts of the ECMAScript lexical grammar. The JSON syntactic grammar describes how sequences of tokens from the JSON lexical grammar can form syntactically correct JSON object descriptions.

Productions of the JSON lexical grammar are distinguished by having two colons "**::**" as separating punctuation. The JSON lexical grammar uses some productions from the ECMAScript lexical grammar. The JSON syntactic grammar is similar to parts of the ECMAScript syntactic grammar. Productions of the JSON syntactic grammar are distinguished by using one colon "**:**" as separating punctuation.

5.1.6 Grammar Notation

Terminal symbols of the lexical, RegExp, and numeric string grammars, and some of the terminal symbols of the other grammars, are shown in **`fixed width`** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a script either exactly as written or using equalvant Unicode escape sequences (see clause 6). All terminal symbol characters specified in this way are to be understood as the appropriate Unicode character from the ASCII range, as opposed to any similar-looking characters from other Unicode ranges.

Deleted: program

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

WhileStatement **:**
    `while (` *Expression* **`)`** *Statement*

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

ArgumentList **:**
    AssignmentExpression
    ArgumentList **`,`** *AssignmentExpression*

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

The subscripted suffix "$_{opt}$", which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

VariableDeclaration **:**
    Identifier Initialiser$_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**
    Identifier
    Identifier Initialiser

and that:

IterationStatement **:**
    `for` **`(`** *ExpressionNoIn*$_{opt}$ **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*

is a convenient abbreviation for:

IterationStatement **:**
    `for` **`(`** **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
    `for` **`(`** **`;`** **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for` **`(`** **`;`** *Expression* **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** **`;`** *Expression*$_{opt}$ **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** *Expression* **`;`** *Expression*$_{opt}$ **`)`** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
    `for` **`(`** **`;`** **`;`** **`)`** *Statement*
    `for` **`(`** **`;`** **`;`** *Expression* **`)`** *Statement*
    `for` **`(`** **`;`** *Expression* **`;`** **`)`** *Statement*
    `for` **`(`** **`;`** *Expression* **`;`** *Expression* **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** **`;`** **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** **`;`** *Expression* **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** *Expression* **`;`** **`)`** *Statement*
    `for` **`(`** *ExpressionNoIn* **`;`** *Expression* **`;`** *Expression* **`)`** *Statement*

so the nonterminal *IterationStatement* actually has eight alternative right-hand sides.

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

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

which is merely a convenient abbreviation for:

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

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

If the phrase "[lookahead ∉ *set*]" appears in the right-hand side of a production, it indicates that the production may not be used if the immediately following input token is a member of the given *set*. The *set* can be written as a list of terminals enclosed in curly braces. For convenience, the set can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand. For example, given the definitions

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

DecimalDigits **::**
DecimalDigit
DecimalDigits DecimalDigit

the definition

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

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

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

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

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

Deleted: program

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.

Deleted: program

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

Deleted: z

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

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

Identifier **::**
IdentifierName **but not** *ReservedWord*

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

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

SourceCharacter **::**
any Unicode character

Deleted: code unit

5.2 Algorithm Conventions

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

Algorithms may be explicitly parameterised, in which case the names and usage of the parameters must be provided as part of the algorithm's definition. In order to facilitate their use in multiple parts of this specification, some algorithms, called *abstract operations*, are named and written in parameterised functional form so that they may be referenced by name from within other algorithms.

Deleted: z

Deleted: the

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 non-terminal symbols of the production alternative as if they were parameters of the algorithm. When used in this manner, non-terminal symbols refer to the actual alternative definition that is matched when parsing the script souce code.

Deleted: program

Unless explicitly specified otherwise, all chain productions have an implicit associated definition for every algorithm that is might be applied to that production's left-hand side nonterminal. The implicit simply reapplies the same algorithm name with the same parameters, if any, to the chain production's sole right-hand side nonterminal and then result. For example, assume there is a production

Deleted: ra

Block **:**
{ *StatementList* **}**

but there is no evalution algorithm that is explicitly specified for that production. If in some algorithm there is a statement of the form: "Return the result of evaluating *Block*" it is implicit that the algorithm has an evalution algorithm of the form:

Runtime Semantics: Evaluation

Block **:** **{** *StatementList* **}**

1. Return the result of evaluating *StatementList*

For clarity of expression, algorithm steps may be subdivided into sequential substeps. Substeps are indented and may themselves be further divided into indented substeps. Outline numbering conventions are used to identify substeps with the first level of substeps labelled with lower case 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:

Deleted: When an algorithm is to produce a value as a result, the directive "return *x*" is used to indicate that the result of the algorithm is the value of *x* and that the algorithm should terminate. The notation Result(*n*) is used as shorthand for "the result of step *n*". ¶

1. Top-level step
    a. Substep.

b. Substep
   i. Subsubstep.
   ii. Subsubstep.
      1. Subsubsubstep
         a Subsubsubsubstep

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

A step may specify the iterative application of its substeps.

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

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

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

Deleted: yields

The mathematical function sign(x) produces 1 if x is positive and −1 if x is negative. The sign function is not used in this standard for cases when x is zero.

Deleted: yields

The mathematical function min(x_1, x_2, ..., x_n) produces the mathematically smallest of x_1 through x_n.

Deleted: yields

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 $x-k = q \times y$ for some integer q.

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

Deleted: yields

NOTE floor(x) = x–(x modulo 1).

5.3 Static Semantic Rules

Deleted: If an algorithm is defined to "throw exception", execution of the algorithm is terminated and no result is returned. The call algorithms are also terminated, until an algorithm step is reached that explicitly deals with the exception, using terminology such as an exception was thrown…". Once such an algorithm step has been encountered the exception is no longer considered to have occurred.¶

Context-free grammars are not sufficiently powerful to express all the rules that define whether a stream of input elements make up a valid ECMAScript script that may be evaluated. In some situations additional rules are needed that may be expressed using either ECMAScript algorithm conventions or prose requirements. Such rules are always associated with a production of a grammar and are called the *static semantics* of the production.

Deleted: program

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.

Deleted: Such algorithms may be parameterized.

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 non-terminal of the grammar that includes the associated production. The default definition of Contains is:

Deleted: grammer

1. For each terminal and non-terminal grammar symbol, *sym*, in the definition of this production do
   a. If *sym* is the same grammar symbol as *symbol*, return **true**.

b. If *sym* is a non-terminal, then
    i. Let *contained* be the result of Contains for *sym* with argument *symbol*.
    ii. If *contained* is **true**, return **true**.
2. Return **false**.

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

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

6 Source Text

Syntax

SourceCharacter **::**
  any Unicode character

The ECMAScript code is expressed using Unicode, version 5.1 or later. ECMAScript source text is a sequence of Unicode characters. The phrase "Unicode character" refers to the abstract linguistic or typographical unit represented by a single Unicode scalar value. The actual encodings used to store and interchange ECMAScript source text is not relevant to this specification. Any well-defined encoding such as UTF-32 or UTF-16 may be used. Source text might even be externally represented using a non-Unicode character encoding. 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 an abstract Unicode character with a corresponding Unicode scalar value. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text.

The phrase "code point" refers to such a Unicode scalar value. "Unicode character" only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual "Unicode characters," even though a user might think of the whole sequence as a single character.

In string literals, regular expression literals, template literals and identifiers, any Unicode characters may also be expressed as a Unicode escape sequence that explicitly express a code point's numeric value. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within other contexts, such an escape sequence contextually contributes one Unicode character.

NOTE ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence `\u000A`, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character `000A` is line feed) and therefore the next Unicode character is not part of the comment. Similarly, if the Unicode escape sequence `\u000A` occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write `\n` instead of `\u000A` to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a Unicode character to the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

ECMAScript String values (8.4) are computational sequences of 16-bit integer values called "code units". ECMAScript language constructs that generate string values from *SourceCharacter* sequences use UTF-16 encoding to generate the code unit values.

Static Semantics: UTF-16 Encoding

The UTF-16 Encoding of a numeric code point value, *cp*, is determined as follows:

1. Assert: $0 \le cp \le 0x10FFFF$
2. If $cp \le 65535$, then return *cp*.

Deleted:
Deleted: l
Deleted: An
Deleted: *Program*
Deleted: static semantic
Deleted: used to
Deleted: a
Deleted: *Program*
Deleted: prior to the first evaluation of that *Program*
Comment [AW2]: Perhaps this should be somewher else. Currently we don't have a section that enumerate all the steps in loading and evaluating a program.
Deleted: static semantic
Deleted: *Program*
Deleted:
Deleted: Static semantic rule violations are early erro (see clause 16) and reported in the same manner as syntax errors.
Deleted: ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 3.0 or later. The text is expected to have been normalised to Unicode Normalization Form (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves. ECMAScript source text is assumed to be a sequence 16-bit code units for the purposes of this specification. Such a source text may include sequences of 16-bit co units that are not valid UTF-16 character encodings. If actual source text is encoded in a form other than 16-b code units it must be processed as if it was first conver to UTF-16.¶
Deleted: code unit
Deleted: well defined
Deleted: source character
Deleted: s
Deleted: Throughout the rest of this document, the phrase "code unit" and the word "character" will be use to refer to a 16-bit unsigned value used to represent a single 16-bit unit of text. The phrase "Unicode characte will be used to refer to the abstract linguistic or
Deleted: quasi
Deleted: (code unit)
Deleted: consisting
Deleted: of six characters, namely `\u` plus four
Deleted: Within a string literal or regular expression
Deleted: the
Deleted: to the identifier
Deleted: ¶
Formatted: Note
Deleted: NOTE Although this document sometimes
Deleted: String value of the
Deleted: deterimined

3. Let *cu1* be floor((*cp* – 65536) / 1024) + 55296. NOTE 55296 is 0xD800.
4. Let *cu2* be ((*cp* – 65536) modulo 1024) + 56320. NOTE 56320 is 0xDC00.
5. Return the code unit sequence consisting of *cu1* followed by *cu2*.

7 Lexical Conventions

The source text of an ECMAScript script is first converted into a sequence of input elements, which are tokens, line terminators, comments, or white space. The source text is scanned from left to right, repeatedly taking the longest possible sequence of characters as the next input element.

There are several situations where the identification of lexical input elements is sensitive to the syntactic grammar context that is consuming the input elements. This requires multiple goal symbols for the lexical grammar. The *InputElementDiv* goal symbol is the default goal symbol and is used in those syntactic grammar contexts where a leading division (`/`) or division-assignment (`/=`) operator is permitted. The *InputElementRegExp* goal symbol is used in all syntactic grammar contexts where a *RegularExpressionLiteral* is permitted. The *InputElementTemplateTail* goal is used in syntactic grammar contexts where a *TemplateLiteral* logically continues after a substitution element.

NOTE There are no syntactic grammar contexts where both a leading division or division-assignment, and a leading *RegularExpressionLiteral* are permitted. This is not affected by semicolon insertion (see 7.9); in examples such as the following:

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

where the first non-whitespace, non-comment character after a *LineTerminator* is slash (`/`) and the syntactic context allows division or division-assignment, no semicolon is inserted at the *LineTerminator*. That is, the above example is interpreted in the same way as:

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

Syntax

InputElementDiv **::**
  WhiteSpace
  LineTerminator
  Comment
  Token
  DivPunctuator
  RightBracePunctuator

InputElementRegExp **::**
  WhiteSpace
  LineTerminator
  Comment
  Token
  RightBracePunctuator
  RegularExpressionLiteral

InputElementTemplateTail **::**
  WhiteSpace
  LineTerminator
  Comment
  Token
  DivPunctuator
  TemplateSubstitutionTail

Deleted: program
Deleted: o
Deleted: There
Deleted: are two
Deleted: other
Deleted: *Quasi*
Deleted: *Quasi*
Comment [AWB93]: May need to also say something about TemplateSubstitution tail. Also need to consider with there are any ASI issues concerning it.
Deleted: *Quasi*
Deleted: *Quasi*

7.1 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.

<ZWNJ> and <ZWJ> are format-control characters that are used to make necessary distinctions when forming words or phrases in certain languages. In ECMAScript source text, <ZWNJ> and <ZWJ> may also be used in an identifier after the first character.

<BOM> is a format-control character used primarily at the start of a text to mark it as Unicode and to allow detection of the text's encoding and byte order. <BOM> characters intended for this purpose can sometimes also appear after the start of a text, for example as a result of concatenating files. <BOM> characters are treated as white space characters (see 7.2).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarised in Table 1.

Table 1 — Format-Control Character Usage

Code Point	*Name*	*Formal Name*	*Usage*
`U+200C`	Zero width non-joiner	<ZWNJ>	*IdentifierPart*
`U+200D`	Zero width joiner	<ZWJ>	*IdentifierPart*
`U+FEFF`	Byte Order Mark	<BOM>	*Whitespace*

7.2 White Space

White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space characters may occur between any two tokens and at the start or end of input. White space characters may occur within a *StringLiteral*, a *RegularExpressionLiteral*, a *Template*, or a *TemplateSubstitutionTail* where they are considered significant characters forming part of a literal value. They may also occur within a *Comment*, but cannot appear within any other kind of token.

The ECMAScript white space characters are listed in Table 2.

Table 2 — Whitespace Characters

Code Point	*Name*	*Formal Name*
`U+0009`	Tab	<TAB>
`U+000B`	Vertical Tab	<VT>
`U+000C`	Form Feed	<FF>
`U+0020`	Space	<SP>
`U+00A0`	No-break space	<NBSP>
`U+FEFF`	Byte Order Mark	<BOM>
Other category “Zs”	Any other Unicode “space separator”	<USP>

ECMAScript implementations must recognise all of the white space characters defined in Unicode 5.1. Later editions of the Unicode Standard may define other white space characters. ECMAScript implementations may recognise white space characters from later editions of the Unicode Standard.

Deleted: , quasis
Deleted: *Unit*
Deleted: *Value*
Deleted: `\u`
Deleted: `\u`
Deleted: `\u`
Deleted: also
Deleted: or
Deleted:
Deleted: *Quasi*
Deleted: *Quasi*
Deleted: (
Deleted: the
Deleted:)
Deleted: or
Deleted: *Unit*
Deleted: *Value*
Deleted: `\u`
Deleted: `\u`
Deleted: `\u`
Deleted: `\u`
Deleted: `\u`
Deleted: `\u`
Deleted: 3
Deleted: 0

Syntax

WhiteSpace **::**
<TAB>
<VT>
<FF>
<SP>
<NBSP>
<BOM>
<USP>

7.3 Line Terminators

Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. Line terminators also affect the process of automatic semicolon insertion (7.9). 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* (7.4) but cannot occur within a *SingleLineComment*.

Line terminators are included in the set of white space characters that are matched by the `\s` class in regular expressions.

The ECMAScript line terminator characters are listed in Table 3.

Table 3 — Line Terminator Characters

Code Point	*Name*	*Formal Name*
U+`000A`	Line Feed	<LF>
U+`000D`	Carriage Return	<CR>
U+`2028`	Line separator	<LS>
U+`2029`	Paragraph separator	<PS>

Only the Unicode characters in Table 3 are treated as line terminators. Other new line or line breaking Unicode characters are treated as white space but not as line terminators. The sequence <CR><LF> is commonly used as a line terminator. It should be considered a single *SourceCharacter* for the purpose of reporting line numbers.

Syntax

LineTerminator **::**
<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence **::**
<LF>
<CR> [lookahead ∉ <LF>]
<LS>
<PS>
<CR> <LF>

7.4 Comments

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

Comment [AWB94]: Need to talk about lin
terminators in Templates

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Unit*

Deleted: *Value*

Deleted: `\u`

Deleted: `\u`

Deleted: `\u`

Deleted: `\u`

Deleted: character

Deleted: character

Because a single-line comment can contain any Unicode character except a *LineTerminator* character, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all characters from the `//` marker to the end of the line. However, the *LineTerminator* at the end of the line is not considered to be part of the single-line comment; it is recognised separately by the lexical grammar and becomes part of the stream of input elements for the syntactic grammar. This point is very important, because it implies that the presence or absence of single-line comments does not affect the process of automatic semicolon insertion (see 7.9).

Comments behave like white space and are discarded except that, if a *MultiLineComment* contains a line terminator character, then the entire comment is considered to be a *LineTerminator* for purposes of parsing by the syntactic grammar.

Syntax

Comment **::**
    MultiLineComment
    SingleLineComment

MultiLineComment **::**
    `/*` *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*

7.5 Tokens

Syntax

Token **::**
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral
    Template

Deleted: *Quasi*

NOTE The *DivPunctuator*, *RegularExpressionLiteral*, *RightBracePunctuator*, and *TemplateSubstitutionTail* productions define tokens, but are not included in the *Token* production.

Deleted: and

Deleted: *Quasi*

7.6 Identifier Names and Identifiers

IdentifierName, *Identifier*, and *ReservedWord* are tokens that are interpreted according to the Default Identifier Syntax given in Unicode Standard Annex #31, Identifier and Pattern Syntax, with some small modifications. *ReservedWord* is is an enumerated subset of *IdentifierName* and *Identifier* is an *IdentifierName* that is not a *ReservedWord* (see 7.6.1). The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The Unicode characters in the specified categories in version 5.1.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations. ECMAScript implementations may recognise identifier characters defined in later editions of the Unicode Standard.

NOTE 1 This standard specifies specific character additions: The dollar sign (`U+0024`) and the underscore (`U+005f`) are permitted anywhere in an *IdentifierName*, and the characters zero width non-joiner (U+200C) and zero width joiner (U+200D) are permitted anywhere after the first character of an *IdentifierName*.

Unicode escape sequences are permitted in an *IdentifierName*, where they contribute a single Unicode character to the *IdentifierName*. The code point of the contributed character is expressed by the *HexDigits* of the *UnicodeEscapeSequence* (see 7.8.6). The \ preceding the *UnicodeEscapeSequence* and the `u` and `{ }` characters, if they appear, do not contribute characters to the *IdentifierName*. A *UnicodeEscapeSequence* cannot be used to put a character into an *IdentifierName* that would otherwise be illegal. In other words, if a \ *UnicodeEscapeSequence* sequence were replaced by the Unicode character it contributes, the result must still be a valid *IdentifierName* that has the exact same sequence of characters as the original *IdentifierName*. All interpretations of *IdentifierName* within this specification are based upon their actual characters regardless of whether or not an escape sequence was used to contribute any particular characters.

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

NOTE 2 If maximal portability is a concern, programmers should only employ the identifier characters that were defined in Unicode 3.0.

Syntax

Identifier **::**
  IdentifierName **but not** *ReservedWord*

IdentifierName **::**
  IdentifierStart
  IdentifierName IdentifierPart

IdentifierStart **::**
  UnicodeIDStart
  $
  _
  **** *UnicodeEscapeSequence*

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

UnicodeIDStart **::**
  any Unicode character with the Unicode property “ID_Start”.

UnicodeIDContinue **::**
  any Unicode character with the Unicode property “ID_Continue”.

Deleted: *Identifer*,
Deleted: Identifier Names
Deleted: grammar given in the “Identifiers” section of chapter 5 of the Unicode standard
Deleted: An
Deleted: both normative and informative
Deleted: categories
Deleted: 3
Formatted: Note
Deleted: ($)
Deleted: (_)
Deleted: also
Deleted: ,
Deleted: as computed by the CV of the
Deleted: 4
Deleted: es
Deleted: a
Deleted: its
Deleted: *UnicodeEscapeSequence*'s CV
Deleted: identifiers
Deleted: The intent is that the incoming sou text has been converted to normalised form C before it reaches the compiler.
Formatted: Note
Deleted: ECMAScript implementations may recognise identifier characters defined in late editions of the Unicode Standard.
Comment [AWB95]: Norbert suggests chaning this to 5.1.0. Would be really be bett for “portablility”?
Deleted: *UnicodeLetter*
Deleted: *IdentifierStart*
Deleted: *Identifier*
Deleted: *UnicodeCombiningMark* *UnicodeDigit* *UnicodeConnectorPunctuation*
Deleted: *UnicodeLetter*
Deleted: in
Deleted: categories
Deleted: Uppercase letter (Lu)
Deleted: , “Lowercase letter (Ll)”, “Titlecas letter (Lt)”, “Modifier letter (Lm)”, “Other lett (Lo)”, or “Letter number (Nl)”
Deleted: *Start*
Deleted: *UnicodeCombiningMark*
Deleted: in
Deleted: categories
Deleted: Non-spacing mark (Mn)
Deleted: or “Combining spacing mark (Mc

The definitions of the nonterminal *UnicodeEscapeSequence* is given in 7.8.6

Static Semantics: StringValue

Identifier **::** *IdentifierName* **but not** *ReservedWord*

1. Return the StringValue of *IdentifierName*.

IdentifierName **::**
IdentifierStart
IdentifierName IdentifierPart

1. Return the String value consisting of the sequence of code units corresponding to *IdentifierName*. In determining the sequence any occurrences of \ *UnicodeEscapeSequence* are first replaced with the code point represented by the *UnicodeEscapeSequence* and then the code points of the entire *IdentifierName* are converted to code units by UTF-16 Encoding (clause 6) each code point.

7.6.1 Reserved Words

A reserved word is an *IdentifierName* that cannot be used as an *Identifier*.

Syntax

ReservedWord **::**
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

The *ReservedWord* definitions are specified as literal sequences of Unicode characters. However, any Unicode character in a *ReservedWord* can also be expressed by a \ *UnicodeEscapeSequence* that expresses that same Unicode character's code point. Use of such escape sequences does not change the meaning of the *ReservedWord*.

7.6.1.1 Keywords

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

Syntax

Keyword :: **one of**

`break`	`delete`	`import`	`this`
`case`	`do`	`in`	`throw`
`catch`	`else`	`instanceof`	`try`
`class`	`export`	`let`	`typeof`
`continue`	`finally`	`new`	`var`
`const`	`for`	`return`	`void`
`debugger`	`function`	`super`	`while`
`default`	`if`	`switch`	`with`

7.6.1.2 Future Reserved Words

The following words are used as keywords in proposed extensions and are therefore reserved to allow for the possibility of future adoption of those extensions.

Deleted: *UnicodeDigit* **::**¶
any character in the Unicode category "Decimal number (Nd)"¶
UnicodeConnectorPunctuation **::**¶
any character in the Unicode category "Connector punctuation (Pc)"¶

Deleted: 4

Deleted:

Deleted: Semantics¶
The String value of the production

Deleted: is determined as follows:

Deleted:

Deleted: v

Deleted: consisting

Deleted: *The String value of* IdentifierName *is determin as follows:*

Deleted: characters

Deleted: a

Deleted: and

Syntax

FutureReservedWord **:: one of**

		`enum`	`extends`	

The following tokens are also considered to be *FutureReservedWords* when they occur within strict mode code (see 10.1.1). The occurrence of any of these tokens within strict mode code in any context where the occurrence of a *FutureReservedWord* would produce an error must also produce an equivalent error:

`implements`		`private`	`public`	`yield`
`interface`	`package`	`protected`	`static`	

7.7 Punctuators

Syntax

Punctuator **:: one of**

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

DivPunctuator **:: one of**

`/`	`/=`

RightBracePunctuator **::**
`}`

7.8 Literals

7.8.1 Null Literals

Syntax

NullLiteral **::**
`null`

7.8.2 Boolean Literals

Syntax

BooleanLiteral **::**
`true`
`false`

Comment [AWB86]: It isn't clear that exter actually needs to be reserved. It's only usag highly contextual.

Deleted: `class`

Deleted: `super`

Deleted: `const`

Deleted: `export`

Deleted: `import`

Comment [AWB87]: Move to keywords

Deleted: `let`

Deleted: `}`

Deleted: <#>Syntax¶
<#>Literal ::¶
<#>NullLiteral
ValueLiteral
¶
<#>ValueLiteral ::¶
<#>BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral¶

Deleted: <#>Semantics¶
<#>The value of the null literal `null` is the sole value of the Null type, namely n

7.8.3 Numeric Literals

Syntax

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` *BinaryDigit*
`0B` *BinaryDigit*
BinaryIntegerLiteral BinaryDigit

BinaryDigit **:: one of**
`0 1`

OctalIntegerLiteral **::**
`0o` *OctalDigit*
`0O` *OctalDigit*
OctalIntegerLiteral OctalDigit

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

HexIntegerLiteral **::**
`0x` *HexDigits*
`0X` *HexDigits*

Deleted: <#>Semantics¶
<#>The value of the Boolean literal `true` is a va of the Boolean type, namely true.¶
<#>The value of the Boolean literal `false` is a value of the Boolean type, namely false.¶

Comment [AWB78]: From March 29 meeting notes:
Hex floating point literals:
Waldemar: Other languages include these things.
They're rarely used
but when you want one, you really want one. Use case are similar to
that of hex literals.
Will explore adding them.
MarkM: 0x3.p1 currently evaluates to undefined. This would be a
breaking change.
Waldemar: Not clear anyone would notice. How did other languages
deal with this?

Comment [AWB79]: The various Digit productions could be refactored to have less redundency

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

Deleted:
Deleted: *IntegerLiteral*

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

Deleted: *source character*

NOTE For example:

```
3in
```

is an error and not the two input elements `3` and `in`.

A conforming implementation, when processing strict mode code (see 10.1.1), must not extend the syntax of *NumericLiteral* to include *OctalIntegerLiteral* as described in B.1.1.

Static Semantics: MV's

A numeric literal stands for a value of the Number type. This value is determined in two steps: first, a mathematical value (MV) is derived from the literal; second, this mathematical value is rounded as described below.

- The MV of *NumericLiteral* **::** *DecimalLiteral* is the MV of *DecimalLiteral*.
- The MV of *NumericLiteral* **::** *BinaryIntegerLiteral* is the MV of *BinaryIntegerLiteral*.
- The MV of *NumericLiteral* **::** *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- The MV of *NumericLiteral* **::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* **.** is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* **.** *DecimalDigits* is the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n}), where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* **.** *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* **.** *DecimalDigits ExponentPart* is (the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n})) times 10^{e}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* **:: .** *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* **:: .** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* **::** *DecimalIntegerLiteral ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *DecimalIntegerLiteral* **:: 0** is 0.
- The MV of *DecimalIntegerLiteral* **::** *NonZeroDigit* is the MV of *NonZeroDigit*.
- The MV of *DecimalIntegerLiteral* **::** *NonZeroDigit DecimalDigits* is (the MV of *NonZeroDigit* times 10^{n}) plus the MV of *DecimalDigits*, where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalDigits* **::** *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* **::** *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* **::** *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* **::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:: +** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:: -** *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* **:: 0** or of *HexDigit* **:: 0** or of *OctalDigit* **:: 0** or of *BinaryDigit* **:: 0** is 0.
- The MV of *DecimalDigit* **:: 1** or of *NonZeroDigit* **:: 1** or of *HexDigit* **:: 1** or of *OctalDigit* **:: 1** or of *BinaryDigit* **:: 1** is 1.

Deleted:

- The MV of *DecimalDigit* :: **2** or of *NonZeroDigit* :: **2** or of *HexDigit* :: **2** or of *OctalDigit* :: **2** is 2.
- The MV of *DecimalDigit* :: **3** or of *NonZeroDigit* :: **3** or of *HexDigit* :: **3** or of *OctalDigit* :: **3** is 3.
- The MV of *DecimalDigit* :: **4** or of *NonZeroDigit* :: **4** or of *HexDigit* :: **4** or of *OctalDigit* :: **4** is 4.
- The MV of *DecimalDigit* :: **5** or of *NonZeroDigit* :: **5** or of *HexDigit* :: **5** or of *OctalDigit* :: **5** is 5.
- The MV of *DecimalDigit* :: **6** or of *NonZeroDigit* :: **6** or of *HexDigit* :: **6** or of *OctalDigit* :: **6** is 6.
- The MV of *DecimalDigit* :: **7** or of *NonZeroDigit* :: **7** or of *HexDigit* :: **7** or of *OctalDigit* :: **7** is 7.
- The MV of *DecimalDigit* :: **8** or of *NonZeroDigit* :: **8** or of *HexDigit* :: **8** is 8.
- The MV of *DecimalDigit* :: **9** or of *NonZeroDigit* :: **9** or of *HexDigit* :: **9** is 9.
- The MV of *HexDigit* :: **a** or of *HexDigit* :: **A** is 10.
- The MV of *HexDigit* :: **b** or of *HexDigit* :: **B** is 11.
- The MV of *HexDigit* :: **c** or of *HexDigit* :: **C** is 12.
- The MV of *HexDigit* :: **d** or of *HexDigit* :: **D** is 13.
- The MV of *HexDigit* :: **e** or of *HexDigit* :: **E** is 14.
- The MV of *HexDigit* :: **f** or of *HexDigit* :: **F** is 15.
- The MV of *BinaryIntegerLiteral* :: **0b** *BinaryDigit* is the MV of *BinaryDigit*.
- The MV of *BinaryIntegerLiteral* :: **0B** *BinaryDigit* is the MV of *BinaryDigit*.
- The MV of *BinaryIntegerLiteral* :: *BinaryIntegerLiteral BinaryDigit* is (the MV of *BinaryIntegerLiteral* times 2) plus the MV of *BinaryDigit*.
- The MV of *OctalIntegerLiteral* :: **0o** *OctalDigit* is the MV of *OctalDigit*.
- The MV of *OctalIntegerLiteral* :: **0O** *OctalDigit* is the MV of *OctalDigit*.
- The MV of *OctalIntegerLiteral* :: *OctalIntegerLiteral OctalDigit* is (the MV of *OctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.
- The MV of *HexIntegerLiteral* :: **0x** *HexDigits* is the MV of *HexDigits*.
- The MV of *HexIntegerLiteral* :: **0X** *HexDigits* is the MV of *HexDigits*.
- The MV of *HexDigits* :: *HexDigit* is the MV of *HexDigit*.
- The MV of *HexDigits* :: *HexDigits HexDigit* is (the MV of *HexDigits* times 16) plus the MV of *HexDigit*.

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 8.5), 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.

7.8.4 Regular Expression Literals

NOTE A regular expression literal is an input element that is converted to a RegExp object (see 15.10) each time the literal is evaluated. Two regular expression literals in a program evaluate to regular expression objects that never compare as **===** to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by **new RegExp** (see 15.10.4) or calling the **RegExp** constructor as a function (15.10.3).

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 code comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are subsequently parsed using the more stringent ECMAScript Regular Expression grammar (15.10.1).

An implementation may extend the ECMAScript Regular Expression grammar defined in 15.10.1, but it must not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions defined below or the productions used by these productions.

Deleted: <#>The MV of *NumericLiteral* :: *BinaryIntegerLiteral* is the MV of *BinaryIntegerLiteral*.¶
<#>The MV of *BinaryDigit* :: **0** is 0.¶
<#>The MV of *BinaryDigit* :: **1** is 1.¶
<#>The MV of *BinaryIntegerLiteral* :: **0b** *BinaryDigit* ... the MV of *BinaryDigit*.¶
<#>The MV of *BinaryIntegerLiteral* :: **0B** *BinaryDigit* ... the MV of *BinaryDigit*.¶
<#>The MV of *BinaryIntegerLiteral* :: *BinaryIntegerLit*... *BinaryDigit* is (the MV of *BinaryIntegerLiteral* times 2) plus the MV of *BinaryDigit*.¶
<#>The MV of *NumericLiteral* :: *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.¶
<#>The MV of *OctalDigit* :: **0** is 0.¶
<#>The MV of *OctalDigit* :: **1** is 1.¶
<#>The MV of *OctalDigit* :: **2** is 2.¶
<#>The MV of *OctalDigit* :: **3** is 3.¶
<#>The MV of *OctalDigit* :: **4** is 4.¶
<#>The MV of *OctalDigit* :: **5** is 5.¶
<#>The MV of *OctalDigit* :: **6** is 6.¶
<#>The MV of *OctalDigit* :: **7** is 7.¶
<#>The MV of *OctalIntegerLiteral* :: **0o** *OctalDigit* is t... MV of *OctalDigit*.¶
<#>The MV of *OctalIntegerLiteral* :: **0O** *OctalDigit* is t... MV of *OctalDigit*.¶
<#>The MV of *OctalIntegerLiteral* :: *OctalIntegerLiter*... *OctalDigit* is (the MV of *OctalIntegerLiteral* times 8) plu... the MV of *OctalDigit*.¶

Deleted: *HexIntegerLiteral*

Deleted: *HexIntegerLiteral*

Deleted: *HexIntegerLiteral*

Deleted: <#>A conforming implementation, wh... processing strict mode code (see 10.1.1), must n... extend the syntax of *NumericLiteral* to include *OctalIntegerLiteral* as described in B.1.1.¶

Deleted: <#>7.8.4 String Literals¶
NOTE A string literal is zero or more Unicode charact... code points enclosed in single or double quotes. Unico... Each characterscode points may also be be represent... by an escape sequence. All characters may appear literally in a string literal except for the closing quote character, backslash, carriage return, line separator, paragraph separator, and line feed. Any character may appear in the form of an escape sequence. String litera... evaluate to ECAMScript String values. When generati... these string values Unicode characterscode points are UTF-16 encoded as defined in clause 6. Code points belonging to Basic Multilingual Plane are encodedd as single code unit element of the string. All other Unicod... characterscode points are encodedd using UTF-16 as two code unit elements of the string.¶
Syntax¶
StringLiteral **::**¶
" *DoubleStringCharacters*$_{opt}$ **"**
' *SingleStringCharacters*$_{opt}$ **'** ¶
DoubleStringCharacters **::**¶
DoubleStringCharacter DoubleStringCharacters$_{opt}$¶

Formatted: Note

Deleted: Strings of characters

Deleted: passed uninterpreted to the regular expressi... constructor, which interprets them according to its own grammar.

Deleted: regular expression constructor's grammar

Syntax

RegularExpressionLiteral **::**
  / *RegularExpressionBody* **/** *RegularExpressionFlags*

RegularExpressionBody **::**
  RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars **::**
  [empty]
  RegularExpressionChars RegularExpressionChar

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

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

RegularExpressionBackslashSequence **::**
  `\` *RegularExpressionNonTerminator*

RegularExpressionNonTerminator **::**
  SourceCharacter **but not** *LineTerminator*

RegularExpressionClass **::**
  `[` *RegularExpressionClassChars* `]`

RegularExpressionClassChars **::**
  [empty]
  RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar **::**
  RegularExpressionNonTerminator **but not one of** `]` **or** `\`
  RegularExpressionBackslashSequence

RegularExpressionFlags **::**
  [empty]
  RegularExpressionFlags IdentifierPart

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters `//` start a single-line comment. To specify an empty regular expression, use: `/(?:)/`.

Static Semantics: Early Errors

RegularExpressionFlags **::** *RegularExpressionFlags IdentifierPart*

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

Static Semantics: BodyText

RegularExpressionLiteral **::** **/** *RegularExpressionBody* **/** *RegularExpressionFlags*

1. Return the source code that was recognised as *RegularExpressionBody*.

Static Semantics: FlagText

RegularExpressionLiteral **::** **/** *RegularExpressionBody* **/** *RegularExpressionFlags*

Deleted: z

1. Return the source code that was recognised as *RegularExpressionFlags*.

7.8.5 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 **::**
  SourceCharacter **but not one of `** **or \ or $**
  $ [lookahead ∉ **{**]
  **** *EscapeSequence*
  LineContinuation

Static Semantics: TV's and TRV's

A template literal component is interpreted as a sequence of Unicode characters. The Template Value (TV) of a literal component is described in terms of code unit values (CV, 7.8.4) contributed by the various parts of the template literal component. As part of this process, some Unicode characters within the template component are interpreted as having a mathematical value (MV, 7.8.3). In determining a TV, escape sequences are replaced by the code unit of the Unicode characters 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* is the TV of *TemplateCharacter*.
- 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.

Deleted: z

Deleted: <#>Semantics¶
<#>A regular expression literal evaluates to a value of the Object type that is an instance of the standard built-in constructor RegExp. This value determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpret into two Strings Pattern and Flags, respectively. Then each time the literal is evaluated, a new object is created as if by the expression `new RegExp(Pattern, Flags)` where RegExp is the standard built-in constructor with that name. The newly constructed object becomes the value of t *RegularExpressionLiteral*. If the call to `new RegExp` would generate an error as specified in 15.10.4.1 the error must be treated as an early error (Claus 16).¶

Deleted: Quasi

Deleted:

Deleted: *Quasi*

Deleted: *FullQuasi…oSubstitutionTemplate*

Deleted: *FullQuasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi…emplateMiddle*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *Quasi…emplateCharacter Quasi*

Deleted: *Quasi*

Comment [AWB910]: Note that the original proposa allowed $IdentifierName to be used as a substitution without { } around the name.

Deleted: QV…V's and RQV

Deleted: quasi…emplate literal component is interpre

Deleted: QV…V and QRV…RV of *FullQuasi*

Deleted: QV…V and QRV…RV of *Quasi*

Deleted: QV…V and QRV…RV of *Quasi*

Deleted: *r*

Deleted: QV…V and QRV…RV of *Quasi*

Deleted: QV…V of *FullQuasi…oSubstitutionTemplate* :

Deleted: QV…V of *Quasi…emplateHead* **:: `**

Deleted: QV…V of *Quasi…emplateMiddle* **:: }**

Deleted: QV…V of *Quasi…emplateTail* **:: }**

Deleted: QV…V of *Quasi…emplateCharacters* **::**

Deleted: QV…V of *Quasi…emplateCharacters* **::**

- The TV of *TemplateCharacter* :: *SourceCharacter* **but not one of `` ` `` or `\` or `$`** is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- The TV of *TemplateCharacter* :: `$` [lookahead ∉ `{`] is the code unit value 0x0024.
- The TV of *TemplateCharacter* :: `\` *EscapeSequence* is the CV of *EscapeSequence*.
- The TV of *TemplateCharacter* :: *LineContinuation* is the TV of *LineContinuation*.
- The TV of *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* is the TRV of *TemplateCharacter*.
- The TRV of *TemplateCharacters* :: *TemplateCharacter TemplateCharacters* is a sequence consisting of the code units in the TRV of *TemplateCharacter* followed by all the code units in the TRV of *TemplateCharacters*, in order.
- The TRV of *TemplateCharacter* :: *SourceCharacter* **but not one of `` ` `` or `\` or `$`** is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- The TRV of *TemplateCharacter* :: `$` [lookahead ∉ `{`] 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 *EscapeSequence* :: *CharacterEscapeSequence* is the TRV of the *CharacterEscapeSequence*.
- The TRV of *EscapeSequence* :: `0` [lookahead ∉ *DecimalDigit*] is the code unit value 0x0030.
- The TRV of *EscapeSequence* :: *HexEscapeSequence* is the TRV of the *HexEscapeSequence*.
- The TRV of *EscapeSequence* :: *UnicodeEscapeSequence* is the TRV of the *UnicodeEscapeSequence*.
- The TRV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the TRV of the *SingleEscapeCharacter*.
- The TRV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The TRV of *SingleEscapeCharacter* :: **one of** `'` `"` `\` `b` `f` `n` `r` `t` `v` is the CV of the *SourceCharacter* that is that single character.
- The TRV of *HexEscapeSequence* :: `x` *HexDigit HexDigit* is the sequence consisting of code unit value 0x0078 followed by TRV of the first *HexDigit* followed by the TRV of the second *HexDigit*.
- The TRV of *UnicodeEscapeSequence* :: `u` *HexDigit HexDigit HexDigit HexDigit* is the sequence consisting of code unit value 0x0075 followed by TRV of the first *HexDigit* followed by the TRV of the second *HexDigit* followed by TRV of the third *HexDigit* followed by the TRV of the fourth *HexDigit*.
- The TRV of *UnicodeEscapeSequence* :: `u{` *HexDigits* `}` is the sequence consisting of code unit value 0x0075 followed by code unit value 0x007B followed by TRV of *HexDigits* followed by code unit value 0x007D.
- The TRV of *HexDigits* :: *HexDigit* is the TRV of *HexDigit*.
- The TRV of *HexDigits* :: *HexDigits HexDigit* is the sequence consisting of TRV of *HexDigits* followed by TRV of *HexDigit*.
- The TRV of a *HexDigit* is the CV of the *SourceCharacter* that is that *HexDigit*.
- The TRV of *LineContinuation* :: `\` *LineTerminatorSequence* is the sequence consisting of the code unit value 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> [lookahead ∉ <LF>] is the code unit value 0x000D.
- 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 0x000D followed by the code unit value 0x000A.

NOTE TV excludes the code units of *LineContinuation* while TRV includes them.

Deleted: QV…V of *Quasi*
Deleted: QV…V of *Quasi*
Deleted: QV…V of *Quasi*
Deleted: QV…V of *Quasi…emplateCharacte*
Deleted: QV
Deleted: QRV…RV of
Deleted: QRV…RV of *Quasi…emplateHead*
Deleted: QRV…RV of *Quasi…emplateMiddl*
Deleted: QRV…RV of *Quasi…emplateTail* :
Deleted: QRV…RV of
Deleted: QRV…RV of
Deleted: QRV…RV of *Quasi*
Deleted: QRV…RV of *Quasi*
Deleted: QRV…RV of
Deleted: QRV…RV of
Deleted: QRV…RV of *EscapeSequence* ::
Deleted: QRV
Deleted: QRV…RV of *EscapeSequence* ::
Deleted: C…RV of *EscapeSequence* ::
Deleted: QRV…RV of
Deleted: QRV
Deleted: QRV
Deleted: QRV…RV of *HexEscapeSequence* :
Deleted: QRV…RV of *UnicodeEscapeSeque*
Deleted: QRV…RV of *UnicodeEscapeSeque*
Deleted: QRV…RV of *HexDigits* :: *HexDigi*
Deleted: QRV…RV of *HexDigits* :: *HexDigi*
Deleted: QRV
Deleted: QRV…RV of *LineContinuation* :: \
Deleted: QRV
Deleted: QRV
Deleted: QRV
Deleted: QRV
Deleted: QRV
Deleted: QV…V excludes the code units of

7.8.6 String Literals

NOTE A string literal is zero or more Unicode code points enclosed in single or double quotes. Unicode code points may also be represented by an escape sequence. All characters may appear literally in a string literal except for the closing quote character, backslash, carriage return, line separator, paragraph separator, and line feed. Any character may appear in the form of an escape sequence. String literals evaluate to ECMAScript String values. When generating these string values Unicode code points are UTF-16 encoded as defined in clause 6. Code points belonging to Basic Multilingual Plane are encoded as a single code unit element of the string. All other code points are encoded as two code unit elements of the string.

Deleted: be

Syntax

StringLiteral **::**
  " *DoubleStringCharacters*$_{opt}$ **"**
  ' *SingleStringCharacters*$_{opt}$ **'**

DoubleStringCharacters **::**
  DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::**
  SingleStringCharacter SingleStringCharacters$_{opt}$

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

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

LineContinuation **::**
  **** *LineTerminatorSequence*

EscapeSequence **::**
  CharacterEscapeSequence
  0 [lookahead ∉ *DecimalDigit*]
  HexEscapeSequence
  UnicodeEscapeSequence

A conforming implementation, when processing strict mode code (see 10.1.1), must not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.2.

CharacterEscapeSequence **::**
  SingleEscapeCharacter
  NonEscapeCharacter

SingleEscapeCharacter **::** **one of**
  `'  "  \  b  f  n  r  t  v`

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

EscapeCharacter **::**
  SingleEscapeCharacter
  DecimalDigit
  x
  u

HexEscapeSequence **::**
  `x` *HexDigit HexDigit*

UnicodeEscapeSequence **::**
  `u` *HexDigit HexDigit HexDigit HexDigit*
  `u{` *HexDigits* **`}`**

The definition of the nonterminal *HexDigit* is given in 7.8.3. *SourceCharacter* is defined in clause 6.

NOTE A line terminator character cannot appear in a string literal, except as part of a *LineContinuation* to produce the empty character sequence. The correct way to cause a line terminator character to be part of the String value of a string literal is to use an escape sequence such as **`\n`** or **`\u000A`**.

Static Semantics

Static Semantics: Early Errors

UnicodeEscapeSequence **::** **`u{`** *HexDigits* **`}`**

- It is a Syntax Error if the MV of *HexDigits* > 1114111.

Static Semantics: SV's and CV's

A string literal stands for a value of the String type. The String value (SV) of the literal is described in terms of code unit values (CV) contributed by the various parts of the string literal. As part of this process, some Unicode characters within the string literal are interpreted as having a mathematical value (MV), as described below or in 7.8.3.

- 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 CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* **::** *DoubleStringCharacter DoubleStringCharacters* is a sequence of one or two code units that is the CV of *DoubleStringCharacter* followed by all the code units in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter* is a sequence of one or two code units that is the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter SingleStringCharacters* is a sequence of one or two code units that is the CV of *SingleStringCharacter* followed by all the code units in the SV of *SingleStringCharacters* in order.
- The SV of *LineContinuation* **::** **`\`** *LineTerminatorSequence* is the empty code unit sequence.
- The CV of *DoubleStringCharacter* **::** *SourceCharacter* **but not one of `"` or `\` or** *LineTerminator* is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- The CV of *DoubleStringCharacter* **::** **`\`** *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *DoubleStringCharacter* **::** *LineContinuation* is the empty character sequence.
- The CV of *SingleStringCharacter* **::** *SourceCharacter* **but not one of `'` or `\` or** *LineTerminator* is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter* .
- The CV of *SingleStringCharacter* **::** **`\`** *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *SingleStringCharacter* **::** *LineContinuation* is the empty character sequence.
- The CV of *EscapeSequence* **::** *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* **::** **`0`** [lookahead ∉ *DecimalDigit*] is the code unit value 0.
- The CV of *EscapeSequence* **::** *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence* **::** *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.

- The CV of *CharacterEscapeSequence* **::** *SingleEscapeCharacter* is the character whose code unit value is determined by the *SingleEscapeCharacter* according to Table 4:

Table 4 — String Single Character Escape Sequences

Escape Sequence	*Code Unit Value*	*Name*	*Symbol*
`\b`	`0x0008`	backspace	<BS>
`\t`	`0x0009`	horizontal tab	<HT>
`\n`	`0x000A`	line feed (new line)	<LF>
`\v`	`0x000B`	vertical tab	<VT>
`\f`	`0x000C`	form feed	<FF>
`\r`	`0x000D`	carriage return	<CR>
`\"`	`0x0022`	double quote	`"`
`\'`	`0x0027`	single quote	`'`
`\\`	`0x005C`	backslash	`\`

- The CV of *CharacterEscapeSequence* **::** *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* **::** *SourceCharacter* **but not one of** *EscapeCharacter* **or** *LineTerminator* is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter* .
- The CV of *HexEscapeSequence* **:: x** *HexDigit HexDigit* is the code unit value that is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *UnicodeEscapeSequence* **:: u** *HexDigit HexDigit HexDigit HexDigit* is the code unit value that is (4096 times the MV of the first *HexDigit*) plus (256 times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.
- The CV of *UnicodeEscapeSequence* **:: u{** *HexDigits* **}** is the UTF-16 Encoding (clause 6) of the MV of *HexDigits*.

7.9 Automatic Semicolon Insertion

Certain ECMAScript statements (empty statement, variable statement, expression statement, **`do-while`** statement, **`continue`** statement, **`break`** statement, **`return`** statement, and **`throw`** statement) must be terminated with semicolons. Such semicolons may always appear explicitly in the source text. For convenience, however, such semicolons may be omitted from the source text in certain situations. These situations are described by saying that semicolons are automatically inserted into the source code token stream in those situations.

7.9.1 Rules of Automatic Semicolon Insertion

There are three basic rules of semicolon insertion:

1. When, as the script is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
   - The offending token is separated from the previous token by at least one *LineTerminator*.
   - The offending token is **`}`**.
2. When, as the script is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript *script*, then a semicolon is automatically inserted at the end of the input stream.
3. When, as the script is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation "[no *LineTerminator* here]" within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated

Deleted: Table 5
Formatted: Font: (Default) Arial, Complex Script Font: Arial
Field Code Changed
Deleted: 5
Deleted: the
Deleted:
Deleted: program
Deleted: program
Deleted: *Program*
Deleted: program

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 12.6.3).

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

PostfixExpression **:**
  LeftHandSideExpression [no *LineTerminator* here] **`++`**
  LeftHandSideExpression [no *LineTerminator* here] **`--`**

ContinueStatement **:**
  `continue` [no *LineTerminator* here] *Identifier* **`;`**

BreakStatement **:**
  `break` [no *LineTerminator* here] *Identifier* **`;`**

ReturnStatement **:**
  `return` [no *LineTerminator* here] *Expression* **`;`**

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

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`**, or **`throw`** token is encountered and a *LineTerminator* is encountered before the next token, a semicolon is automatically inserted after the **`continue`**, **`break`**, **`return`**, or **`throw`** 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 should start on the same line as the **`return`** or **`throw`** token.

An *Identifier* in a **`break`** or **`continue`** statement should be on the same line as the **`break`** or **`continue`** token.

7.9.2 Examples of Automatic Semicolon Insertion

The source

```
{ 1 2 } 3
```

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

```
{ 1
2 } 3
```

is also not a valid ECMAScript sentence, but is transformed by automatic semicolon insertion into the following:

```
{ 1
;2 ;} 3;
```

which is a valid ECMAScript sentence.

The source

```
for (a; b
)
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion because the semicolon is needed for the header of a **for** statement. Automatic semicolon insertion never inserts one of the two semicolons in the header of a **for** statement.

The source

```
return
a + b
```

is transformed by automatic semicolon insertion into the following:

```
return;
a + b;
```

NOTE The expression **a + b** is not treated as a value to be returned by the **return** statement, because a *LineTerminator* separates it from the token **return**.

The source

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

is transformed by automatic semicolon insertion into the following:

```
a = b;
++c;
```

NOTE The token **++** is not treated as a postfix operator applying to the variable **b**, because a *LineTerminator* occurs between **b** and **++**.

The source

```
if (a > b)
else c = d
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion before the **else** token, even though no production of the grammar applies at that point, because an automatically inserted semicolon would then be parsed as an empty statement.

The source

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

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

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

In the circumstance that an assignment statement must begin with a left parenthesis, it is a good idea for the programmer to provide an explicit semicolon at the end of the preceding statement rather than to rely on automatic semicolon insertion.

8 Types

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

Within this specification, the notation "Type(*x*)" is used as shorthand for "the type of *x*" where "type" refers to the ECMAScript language and specification types defined in this clause.

Deleted: An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boole String, Number, and Object.¶
A specification type corresponds to meta-values that ar used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The specification types are Reference List, Completion, Property Descriptor, Property Identifie Lexical Environment, and Environment Record, and D Block. Specification type values are specification artefa that do not necessarily correspond to any specific entit within an ECMAScript implementation. Specification ty 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.¶

8.1 ECMAScript Language Types

An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Number, and Object. An ECMAScript language value is a value that is characterized by an ECMAScript language type.

8.1.1 The Undefined Type

The Undefined type has exactly one value, called **undefined**. Any variable that has not been assigned a value has the value **undefined**.

8.1.2 The Null Type

The Null type has exactly one value, called **null**.

8.1.3 The Boolean Type

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

8.1.4 The String Type

The String type is the set of all finite ordered sequences of zero or more 16-bit unsigned integer values ("elements"). The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the String is treated as a UTF-16 code unit value. Each element is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at index 0, the next element (if any) at index 1, and so on. The length of a String is the number of elements (i.e., 16-bit values) within it. The empty String has length zero and therefore contains no elements.

Where ECMAScript operations interpret String values, each element is interpreted as a single UTF-16 code unit. However, ECMAScript does not place any restrictions or requirements on the sequence of code units in a String value, so they may be ill-formed when interpreted as UTF-16 code unit sequences. Operations that do not interpret String contents treat them as sequences of undifferentiated 16-bit unsigned integers. No operations ensure that Strings are in a normalized form. Only operations that are explicitly specified to be language or locale sensitive produce language-sensitive results

NOTE The rationale behind this design was to keep the implementation of Strings as simple and high-performing as possible. If ECMAScript source code is in Normalised Form C, string literals are guaranteed to also be normalised, as long as they do not contain any Unicode escape sequences.

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

- A code unit in the range 0 to 0xD7FF or in the range 0xE000 to 0xFFFF is interpreted as a code point with the same value.
- A sequence of two code units, where the first code unit *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) × 0x400 + (*c2* – 0xDC00) + 0x10000.
- A code unit that is in the range 0xD800 to 0xDFFF, but is not part of a surrogate pair, is interpreted as a code point with the same value.

8.1.5 The Number Type

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

Deleted: Unicode

Deleted: (see Clause 6)

Deleted: position

Deleted: position

Deleted: <#>When a String contains act textual data, each element is considered to be a single UTF-16 code unit. Whether or this is the actual storage format of a String the characters within a String are numbere by their initial code unit element position as though they were represented using UTF-1 All operations on Strings (except as otherwise stated) treat them as sequences undifferentiated 16-bit unsigned integers; they do not ensure the resulting String is in normalised form, nor do they ensure language-sensitive results.¶

Deleted: The intent is that textual data comi into the execution environment from outside (e.g., user input, text read from a file or receiv over the network, etc.) be converted to Unico Normalised Form C before the running progra sees it. Usually this would occur at the same time incoming text is converted from its origin character encoding to Unicode (and would impose no additional overhead). Since it is recommended that

Deleted: be

Deleted: (if source text is guaranteed to be normalised) between various Not-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all NaN values are indistinguishable from each other.

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

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

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

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

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

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

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

The remaining 9007199254740990 (that is, $2^{53}-2$) values are denormalised, 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 "round to nearest" mode.)

Some ECMAScript operators deal only with integers in the range -2^{31} through $2^{31}-1$, inclusive, or in the range 0 through $2^{32}-1$, inclusive. These operators accept any value of the Number type but first convert each such value to one of 2^{32} integer values. See the descriptions of the ToInt32 and ToUint32 operators in 9.5 and 9.6, respectively.

8.1.6 The Object Type

An Object is logically a collection of properties. Each property is either a data property, or an accessor property.

Deleted: named

Deleted: named

Deleted: , or an internal property

- A *data property* associates a key value with an ECMAScript language value and a set of Boolean attributes.
- A *accessor property* associates a key value with one or two accessor functions, and a set of Boolean attributes. The accessor functions are used to store or retrieve an ECMAScript language value that is associated with the property.

Properties are identified using key values. A key value is either an ECMAScript String value or a Exotic Symbol object.

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.

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.

8.1.6.1 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 5.

Table 5 — Attributes of a Data Property

Attribute Name	***Value Domain***	***Description***
[[Value]]	Any ECMAScript language type	The value retrieved by a get access of the property.
[[Writable]]	Boolean	If **false**, attempts by ECMAScript code to change the property's [[Value]] attribute using [[Set]] will not succeed.
[[Enumerable]]	Boolean	If **true**, the property will be enumerated by a for-in enumeration (see 12.6.4). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If **false**, attempts to delete the property, change the property to be an accessor property, or change its attributes (other than [[Value]], or changing [[Writable]] to **false**) will fail.

An accessor property associates a key value with the attributes listed in Table 6.

Deleted: *named*
Deleted: name
Deleted: *named*
Deleted: name
Deleted: y
Deleted: n
Deleted: ECMAScript
Comment [AWB1211]: TODO, section reference
Deleted: (??.??.??)8.4.3)
Deleted: An *internal property* has no name and is not directly accessible via ECMAScri language operators. Internal properties exis purely for specification purposes. ¶
Deleted: named (non-internal)
Deleted: *put*
Deleted: accessable
Deleted: properites
Deleted: objacts
Deleted: semanjtics
Deleted: *object* are
Deleted: ,
Deleted: named
Deleted: named
Deleted: name
Deleted: Named
Deleted: reading
Deleted: Put
Deleted: P
Deleted: or [[DefineOwnProperty]]
Deleted: for
Deleted: named
Deleted: name

Table 6 — Attributes of an Accessor Property

Attribute Name	Value Domain	Description
[[Get]]	Object *or* Undefined	If the value is an Object it must be a function Object. The function's [[Call]] internal method (8.6.2) is called with an empty arguments list to retrieve the property value each time a get access of the property is performed.
[[Set]]	Object *or* Undefined	If the value is an Object it must be a function Object. The function's [[Call]] internal method (8.6.2) is called with an arguments list containing the assigned value as its sole argument each time a set access of the property is performed. The effect of a property's [[Set]] internal method may, but is not required to, have an effect on the value returned by subsequent calls to the property's [[Get]] internal method.
[[Enumerable]]	Boolean	If **true**, the property is to be enumerated by a for-in enumeration (see 12.6.4). Otherwise, the property is said to be non-enumerable.
[[Configurable]]	Boolean	If **false**, attempts to delete the property, change the property to be a data property, or change its attributes will fail.

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in Table 7 is used.

Table 7 — Default Attribute Values

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

8.1.6.2 Object Internal Methods and Internal Data Properties

The actual semantics of ECMAScript objects 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 methods are identified within this specification using names enclosed in double square brackets [[ ]]. Internal method names are polymorphic. This means that different ECMAScript object values may perform different algorithms when a common internal method name is invoked upon them. If, at runtime, the implementation of an algorithm attempts to use an internal method of an object that the object does not support, a **TypeError** exception is thrown.

Internal data properties correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms. Depending upon the specific internal data property such state may consist of values of any ECMAScript language type or of specific ECMA specification type values. Unless explicitly specified otherwise, internal data properties are allocated as part of the process of creating an ECMAScript object and may not be dynamically added to ECMAScript objects. Unless specified otherwise, the initial value of an internal data property is the value **undefined**.

Deleted: Named
Deleted: return
Deleted: P
Deleted: P
Deleted: n
Deleted: is
Deleted: for a named property
Deleted: Properties and
Deleted: This specification uses various internal properties to define the semantics of object values.
Deleted: properties
Deleted: An
Deleted: as if it produced and operated upon internal properties in the manner described here
Deleted: .
Deleted: The names of i
Deleted: properties
Deleted: are
Deleted: When
Deleted: uses an
Deleted: property
Deleted: and the object does not implement the indicated internal property
Deleted: an

Table 8 summarises the essential *internal methods* used by this specification that are applicable to all ECMAScript objects. Every object must have algorithms for all of the essential internal methods. However, all objects do not necessarily use the same algorithms for those methods.

The "Signature" column of Table 8 and other similar tables describes the invocation pattern for each internal method. The invocation pattern always includes a parenthesised list of descriptive parameter names. If a parameter name is the same as an ECMAScript type name then the name describes the required type of the parameter value. If an internal method explicitly returns a value, its parameter list is followed by the symbol "→" and the type name of the returned value. The type names used in signatures refer to the types defined in Clause 8 augmented by the following additional names. "*any*" means the value may be any ECMAScript language type. An internal method implicitly returns a Completion Record as described in 8.8. In addition to its parameters, an internal method always has access to the object upon which it is invoked as a method.

Deleted: Table 8

Deleted:

Deleted: *properties*

Deleted: Value Type Domain

Deleted: s

Deleted: the following

Deleted: define

Deleted: the types of values associated with internal properties

Deleted: The type names refer to the types defined in Clause 8 augmented by the followi additional names. "*any*" means the value ma be any ECMAScript language type. "*primitive* means Undefined, Null, Boolean, String, or Number. "*SpecOp*" means the internal prope is an internal method, an implementation provided procedure defined by an abstract operation specification. "SpecOp" is followed a

Deleted: "SpecOp"

Deleted: "*primitive*" means Undefined, Null, Boolean, String, or Number.

Table 8 — Essential Internal Methods

Internal Method	*Signature*	*Description*
[[GetInheritance]]	()→Object or Null	Determine the object that provides inherited properties for this object. A **null** value indicates that there are no inherited properties. an object.
[[SetInheritance]]	(*Object* or Null)→Boolean	Associate with an object another object that provides inherited properties. Passing **null** indicates that there are no inherited properties. Returns **true** indicating that the operation was completed successfully or **false** indicating that the operation was not successful.
[[IsExtensible]]	( )→Boolean	Determine whether it is permitted to add additional properties to an object.
[[PreventExtensions]]	( )→Boolean	Control whether new properties may be added to an object. Returns **true** indicating that the operation was completed successfully or **false** indicating that the operation was not successful.
[[HasOwnProperty]]	(*propertyKey*) → Boolean	Returns a Boolean value indicating whether the object already has an own property whose key is *propertyKey*.
[[GetOwnProperty]]	(*propertyKey*) → Undefined or Property Descriptor	Returns a Property Descriptor for the own property of this object whose key is *propertyKey*, or **undefined** if no such property exists.
[[HasProperty]]	(*propertyKey*) → Boolean	Returns a Boolean value indicating whether the object already has either an own or inherited property whose key is *propertyKey*.
[[Get]]	(*propertyKey*, *Receiver*) → *any*	Retrive the value of an object's property using the *propertyKey* parameter. If any ECMAScript code must be executed to retrieve the property value, *Receiver* is used as the **this** value when evaluating the code.
[[Set]]	(*propertyKey*,*value*, *Receiver*) → *Boolean*	Try to set the value of an object's property indentified by *propertyKey* to *value*. If any ECMAScript code must be executed to set the property value, *Receiver* is used as the **this** value when evaluating the code. Returns **true** indicating that the property value was set or **false** indicating that it could not be set.
[[Invoke]]	(*propertyKey*, a List of *any*, *Receiver*) → *any*	Retrive the value of an object's property using the *propertyKey* parameter. If the retrieved property value is a function, [[Call]] it using the List as the arguments list and *Receiver* as the **this** value. A **TypeError** is thrown if a function is not retrieved.
[[Delete]]	(*propertyKey*) → Boolean	Removes the own property indentified by the *propertyKey* parameter from the object. Return **false** if the property was not deleted and is still present. Return **true** if the property was deleted or was not present.
[[DefineOwnProperty]]	(*propertyKey*, *PropertyDescriptor*) → Boolean	Creates or alters the named own property to have the state described by a Property Descriptor. Returns **true** indicating that the property was successfully created/updated or **false** indicating that the property could not be created or updated.
[[Enumerate]]	()→Object	Returns an iterator object over the string values of the keys of the enumerable properties of the object.

Deleted: Has

Deleted: Integrity

Deleted: String

Deleted: the property structure of an object is fixed to least the specified level. The argument is one of the values "`nonextensible`", "`sealed`", or "`frozen`"

Deleted: SetIntegrityPp

Deleted: String

Deleted: Restrict the mutability of an object's properti to that which is allowed for the specified integtrrity leve The argument is one of the values "`nonextensible`", "`sealed`", or "`frozen`".

Deleted: The integrity level of an object may be raised but may not be lowered.

Deleted: [[GetP]]

Deleted: (*propertyKey*, *Receiver*) → any

Deleted: Retrive the value of an object's property usin the *propertyKey* parameter. If any ECMAScript code m be executed to retrieve the property value, *Receiver* is used as the **this** value when evaluating the code.

Deleted: P

Deleted: is the property was not deleted because its [[Configurable]] attribute is **false**

Deleted: Otherwise r

Deleted: that

[[OwnPropertyKeys]]	()→Object	Returns an Iterator object that produces all of the own property keys for the object.

Table 9 summarises additional essential internal methods that are supported by objects that may be called as functions.

Table 9 — Additional Essential Internal Methods of Function Objects

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

8.1.6.3 Invariants of the Essential Internal Methods

Current this section is just a bunch of material merged together from the ES5 spec. and from the wiki Proxy pages. It need to be completely reworked.

The intent is that it lists all invariants of the Essential Internal Methods. This includes both invariants that are enforced for Proxy objects and other invariants that may not be enfored.

Definitions:

- The *target* of an internal method is the object the internal method is called upon.
- A *sealed property* is a non-configurable own property of a target.
- A *frozen property* is a non-configurable non-writable own property of a target.
- A new property is a property that does not exist on a non-extensible target.
- Two property descriptors *desc1* and *desc2* for a property key value are incompatible if:
  1. *Descl* is produced by calling [[GetOwnPropertyDescriptor]] of *target* with *key*, and
  2. Calling [[DefineOwnProperty]] of *target* with arguments *key* and *desc2* would throw a *TypeError* exception.

Exotic objects may define additional constraints upon their [[Set]] internal method behaviour. If possible, exotic objects should not allow [[Set]] operations in situations where this definition of [[CanPut]] returns **false**.

[[GetInheritance]]

Every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] internal data property must eventually lead to a **null** value).

getOwnPropertyDescriptor

Non-configurability invariant: cannot return incompatible descriptors for sealed properties

Deleted: List of (String or Symbol)
Deleted: Arrayinterator
Deleted: containing
Deleted: except those that are private Symbols
Deleted: [[Freeze]]
Deleted:
Deleted: The
Deleted: Table 9
Deleted: the
Deleted: properties
Deleted: used by this specification that are only applicable to some ECMAScript objects
Deleted: must
Deleted: be
Deleted: all
Deleted: .
Deleted: .
Deleted: *Value Type Domain*
Deleted: or Reference
Deleted: Only callable objects that are host objects may return Reference values.
Deleted: <#>¶
Deleted: P
Comment [AWB1212]: Need to decide wh replaces this when CanPut goes away
Deleted: P

Non-extensibility invariant: must return **undefined** for new properties
Invariant checks:
if trap returns **undefined**, check if the property is configurable
if property exists on target, check if the returned descriptor is compatible
if returned descriptor is non-configurable, check if the property exists on the target and is also non-configurable

defineProperty

Non-configurability invariant: cannot succeed (return true) for incompatible changes to sealed properties
Non-extensibility invariant: must reject (return **false**) for new properties
Invariant checks:
on success, if property exists on target, check if existing descriptor is compatible with argument descriptor
on success, if argument descriptor is non-configurable, check if the property exists on the target and is also non-configurable

getOwnPropertyNames

Non-configurability invariant: must report all sealed properties
Non-extensibility invariant: must not list new property names
Invariant checks:
check whether all sealed target properties are present in the trap result
If the target is non-extensible, check that no new properties are listed in the trap result

deleteProperty

Non-configurability invariant: cannot succeed (return true) for sealed properties
Invariant checks:
on success, check if the target property is configurable

getPrototypeOf

Invariant check: check whether the target's prototype and the trap result are identical (according to the egal operator)

freeze | seal | preventExtensions

Invariant checks:
on success, check if isFrozen(target), isSealed(target) or !isExtensible(target)

isFrozen | isSealed | isExtensible

Invariant check: check whether the boolean trap result is equal to isFrozen(target), isSealed(target) or isExtensible(target)

hasOwn

Non-configurability invariant: cannot return **false** for sealed properties
Non-extensibility invariant: must return **false** for new properties
Invariant checks:
if **false** is returned, check if the target property is configurable
if **false** is returned, the property does not exist on target, and the target is non-extensible, throw a TypeError

has

Non-configurability invariant: cannot return **false** for sealed properties
Invariant checks:
if **false** is returned, check if the target property is configurable

get

Non-configurability invariant: cannot return inconsistent values for frozen data properties, and must return **undefined** for sealed accessors with an **undefined** getter

Invariant checks:

if property exists on target as a data property, check whether the target property's value and the trap result are identical (according to the egal operator)

if property exists on target as an accessor, and the accessor's get attribute is **undefined**, check whether the trap result is also **undefined**.

set

Non-configurability invariant: cannot succeed (return true) for frozen data properties or sealed accessors with an **undefined** setter
Invariant checks:
on success, if property exists on target as a data property, check whether the target property's value and the update value are identical (according to the egal operator)
on success, if property exists on target as an accessor, check whether the accessor's set attribute is not **undefined**

keys

Non-configurability invariant: must report all enumerable sealed properties
Non-extensibility invariant: must not list new property names
Invariant checks:
Check whether all enumerable sealed target properties are listed in the trap result
If the target is non-extensible, check that no new properties are listed in the trap result

enumerate

Non-configurability invariant: must report all enumerable sealed properties
Invariant checks:
Check whether all enumerable sealed target properties are listed in the trap result

~~NOTE This specification defines no ECMAScript language operators or built-in functions that permit a program to modify an object's [[Prototype]] internal properties or to change the value of [[Extensible]] from **false** to **true**. Implementation specific extensions that modify [[Prototype]] or [[Extensible]] must not violate the invariants defined in the preceding paragraph.~~

Unless otherwise specified, the standard ECMAScript objects are ordinary objects and behave as described in 8.3. Some standard objects are exotic objects and have behaviour defined in 8.4.

Exotic objects may implement internal methods in any manner unless specified otherwise; for example, one possibility is that [[Get]] and [[Set]] for a particular exotic object indeed fetch and store property values but

Comment [AWB1213]: These are placeholders based upon the proxy trap invariants. We need to provide new versions all the essential internal methods.

Deleted: Clause 8.12 provides the algorithm that defined the default behaviour for the internal methods in Table 8. An *ordinary obje* is an object that uses all of these default algorithms. An *exotic object* is an object that uses some other algorithm to provide alterna behaviour for one or more of these internal methods. The descriptions in these tables indicate their behaviour for native ECMAScrip objects, unless stated otherwise in this document for particular kinds of native ECMAScript objects. ¶

Deleted: Host objects may support these internal properties with any implementation-dependent behaviour as long as it is consiste with the specific host object restrictions stated this document.¶

Deleted: The "Value Type Domain" columns the following tables define the types of values associated with internal properties. The type names refer to the types defined in Clause 8 augmented by the following additional names "*any*" means the value may be any ECMAScr language type. "*primitive*" means Undefined, Null, Boolean, String, or Number. "*SpecOp*" means the internal property is an internal method, an implementation provided procedu defined by an abstract operation specification "SpecOp" is followed by a list of descriptive parameter names. If a parameter name is the same as a type name then the name describe the type of the parameter. If a "SpecOp" retur a value, its parameter list is followed by the symbol "→" and the type of the returned valu

Comment [AWB1214]: No longer true because of Proxies.

Deleted: ~~Every object (including host objects must implement all of the internal properties~~

Formatted: Strikethrough

Formatted: Strikethrough

Deleted: ~~[[Class]] or~~

Deleted: ~~[[Class]],~~

Deleted: The value of the [[Class]] internal property is defined by this specification for ev

Deleted: common internal methods of native

Deleted: 12

Deleted: Array objects have a slightly differe implementation of the [[DefineOwnProperty]]

Deleted:]

Deleted: .

Deleted: Host

Deleted: these

Deleted: P

Deleted: Put

Deleted: P

Deleted: host

[[HasOwnProperty]] always generates **false**. However, if any specified manipulation of an exotic object's internal properties is not supported by an implementation, that manipulation must throw a **TypeError** exception when attempted.

The [[GetOwnProperty]] internal method of all objects must conform to the following invariants for each property of the object:

- If a property is described as a data property and it may return different values over time, then either or both of the [[Writable]] and [[Configurable]] attributes must be **true** even if no mechanism to change the value is exposed via the other internal methods.

- If a property is described as a data property and its [[Writable]] and [[Configurable]] are both **false**, then the SameValue (according to 9.12) must be returned for the [[Value]] attribute of the property on all calls to [[GetOwnProperty]].

- If the attributes other than [[Writable]] may change over time or if the property might disappear, then the [[Configurable]] attribute must be **true**.

- If the [[Writable]] attribute may change from **false** to **true**, then the [[Configurable]] attribute must be **true**.

- If the result of calling an object's [[IsExtensible]] internal method has been observed by ECMAScript code to be **false**, then if a call to [[GetOwnProperty]] describes a property as non-existent all subsequent calls must also describe that property as non-existent.

The [[DefineOwnProperty]] internal method of all objects must not permit the addition of a new property to an object if the [[Extensible]] internal method of that object has been observed by ECMAScript code to be **false**.

If the result of calling the [[IsExtensible]] internal method of an object has been observed by ECMAScript code to be **false** then it must not subsequently become **true**.

8.1.6.4 Well-Known Symbols and Intrinsics

Well-known symbols are built-in Symbol exotic objects (8.4.4) 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 objects are shared by all Code Realms (10.3) and the value of their [[Private]] attribute is **false**.

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 10.

Deleted: host
Deleted: host
Deleted: host
Deleted: value
Deleted: of
Deleted: the
Deleted: host
Deleted: property
Deleted: host
Deleted: host
Deleted: that
Deleted: property
Deleted: host
Deleted: property
Deleted: that
Deleted: host
Deleted: exotic s
Deleted: objects
Deleted: values
Deleted: 8.4.4
Deleted: values
Deleted: internal data property (forward ref)

Table 10 - Well-known Symbols

Specification Name	Value and Purpose
@@create	A method used to allocate an object. Called from the [[Construct]] internal method.
@@hasInstance	A method that determines if a constructor object recognises an object as one of the constructor's instances. Called by the semantics of the `instanceof` operator.
@@isRegExp	A Boolean value that if true indicates that an object may be used as a regular expression.
@@iterator	A method that returns the default iterator for an object. Called by the semantics of the for-of statement.
@@ToPrimitive	A method that converts an object to a corresponding primitive value. Called by the ToPrimitive abstract operation.
@@toStringTag	A string value that is used in the creation of the default string description of an object. Called by the built-in method Object.prototype.toString.

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 10.4. The well-known intrinsics are listed in Table 11.

Deleted: 10
Deleted: 1010101011
Deleted: z
Deleted: a
Deleted: Code
Deleted: f
Deleted: specific
Deleted: Code
Deleted: Code
Deleted: e
Deleted: Code
Deleted: a
Deleted: 3
Deleted: intrincs

Table 11 — Well-known Intrinsic Objects

Intrinsic Name	ECMAScript Language Association
%Object%	The initial value of the global object property named "Object".
%ObjectPrototype%	The initial value of the "prototype" data property of the intrinsic %Object%.
%ObjProto_toString%	The initial value of the "toString" data property of the intrinsic %ObjectPrototype%.
%Function%	The initial value of the global object property named "Function".
%FunctionPrototype%	The initial value of the "prototype" data property of the intrinsic %Function%.
%Array%	The initial value of the global object property named "Array".
%ArrayPrototype%	The initial value of the "prototype" data property of the intrinsic %Array%.
%ArrayIteratorPrototype%	The prototype object used for Iterator objects created by the CreateArrayIterator abstract operation.
%String%	The initial value of the global object property named "String".
%StringPrototype%	The initial value of the "prototype" data property of the intrinsic %String%.
%Boolean%	The initial value of the global object property named "Boolean".
%BooleanPrototype%	The initial value of the "prototype" data property of the intrinsic %Boolean%.
%Number%	The initial value of the global object property named "Number".
%NumberPrototype%	The initial value of the "prototype" data property of the intrinsic %Number%.
%Date%	The initial value of the global object property named "Date".
%DatePrototype%	The initial value of the "prototype" data property of the intrinsic %Date%.
%RegExp%	The initial value of the global object property named "RegExp".
%RegExpPrototype%	The initial value of the "prototype" data property of the intrinsic %RegExp%.
%Map%	The initial value of the global object

Deleted: 1111111125

Formatted Table

Deleted: interator

Formatted Table

	property named `"Map"`.
%MapPrototype%	The initial value of the `"prototype"` data property of the intrinsic %Map%.
%MapIteratorPrototype%	The prototype object used for Iterator objects created by the CreateMapIterator abstract operation
%WeakMap%	The initial value of the global object property named `"WeakMap"`.
%WeakMapPrototype%	The initial value of the `"prototype"` data property of the intrinsic %WeakMap%.
%Set%	The initial value of the global object property named `"Set"`.
%SetPrototype%	The initial value of the `"prototype"` data property of the intrinsic %Set%.
%WeakSet%	The initial value of the global object property named `"WeakSet"`.
%WeakSetPrototype%	The initial value of the `"prototype"` data property of the intrinsic %WeakWeakSet%.
%SetIteratorPrototype%	The prototype object used for Iterator objects created by the CreateSetIterator abstract operation
%GeneratorFunction%	The initial value of the name "GeneratorFunction" exported from the built-in module "std:iteration".
%Generator%	The initial value of the name "Generator" exported from the built-in module "std:iteration"
%GeneratorPrototype%	The initial value of the `prototype` property of the %Generator% intrinsic
%Error%	
%EvalError%	
%RangeError%	
%ReferenceError%	
%SyntaxError%	
%TypeError%	
%URIError%	
%ErrorPrototype%	
%EvalErrorPrototype%	
%RangeErrorPrototype%	
%ReferenceErrorPrototype%	
%SyntaxErrorPrototype%	
%TypeErrorPrototype%	
%URIErrorPrototype%	
%ArrayBuffer%	

Deleted: interator

Formatted Table

Deleted: interator

Deleted: %StopIteration%

Formatted: Indent: First line: 0.28"

Formatted Table

%ArrayBufferPrototype%	The initial value of the "prototype" data property of the intrinsic %ArrayBuffer%.
%TypedArray%	
%TypedArrayPrototype%	The initial value of the "prototype" data property of the intrinsic %TypedArray%.
%Int8Array%	
%Int8ArrayPrototype%	
%DataView%	
%DataViewPrototype%	
%ThrowTypeError%	A function object that conditionally throws a new instance of %TypeError%
???	

Comment [AWB1615]: TODO add all the other TypedArray view intrinsics

Deleted: e

Comment [AWB1216]: TODO more to comeTODO

Formatted Table

Formatted: Heading 2

8.2 ECMAScript Specification Types

A specification type corresponds to meta-values that are used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The specification types are Reference, List, Completion, Property Descriptor, Lexical Environment, Environment Record, and Data Block. Specification type values are specification artefacts that do not necessarily correspond to any specific entity within an ECMAScript implementation. Specification type values may be used to describe intermediate results of ECMAScript expression evaluation but such values cannot be stored as properties of objects or values of ECMAScript language variables.

Deleted: Property Identifier,

Deleted: ¶

8.2.1 Data Blocks

This section is a placeholder for describing the Data Block internal type. The following material is verbatim from the the Binary Data ES wiki proposal. The material has not yet been reviewed or integrated with the rest of this spec.

This spec introduces a new, spec-internal block datatype, intuitively representing a contiguously allocated block of binary data. Blocks are not ECMAScript language values and appear only in the program store (aka heap).

A block is one of:

- a number-block
- an array-block[t, n]
- a struct-block[t1, ..., tn]

A number-block is one of:

- an unsigned-integer; i.e., one of uint8, uint16, uint32, or uint64
- a signed-integer; i.e., one of int8, int16, int32, or int64
- a floating-point; i.e., one of float32 or float64

A uintk is an integer in the range [0, 2k). An intk is an integer in the range [-2k-1, 2k-1). A floatk is a floating-point number representable as a k-bit IEE754 value.

An array-block[t, n] is an ordered sequence of n blocks of homogeneous block type t. Each element of the array is stored at in independently addressable location in the program store, and multiple Data objects may contain references to the element.

A struct-block[t1, ..., tn] is an ordered sequence of n blocks of heterogeneous types t1 to tn, respectively. Each field of the struct is stored at in independently addressable location in the program store, and multiple Data objects may contain references to the field.

The spec also introduces a datatype of Data objects, which are ECMAScript objects that encapsulate references to block data in the program store. Every Data object has the following properties:

[[Class]] = “Data”

[[Value]] : reference[block] – a reference to a block in the program store

[[DataType]] : reference[Type] – a reference to a Type object describing this object’s data block

8.2.2 The List and Record Specification Type

The List type is used to explain the evaluation of argument lists (see 11.2.4) in `new` expressions, in function calls, and in other algorithms where a simple list of values is needed. Values of the List type are simply ordered sequences of values. These sequences may be of any length.

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

For notational convenience within this specification, an object literal-like syntax can be used to express a Record value. For example, {[[field1]]: 42, [[field2]]: **false**, [[field3]]: **empty**} defines a Record value that has three fields each of which is initialised to a specific value. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

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

Schema for commonly used Record field combinations may be named, and that name may be used as a prefix to a literal Record value to identify the specific kind of aggregations that is being described. For example: Property Descriptor {[[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true**}.

8.2.3 The Completion Record Specification Type

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

Values of the Completion type are Record values whose fields are defined as by Table 12.

Table 12 — Completion Record Fields

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

The term "abrupt completion" refers to any completion with a [[type]] value other than **normal**.

8.2.3.1 NormalCompletion

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

1. Return NormalCompletion(*argument*).

Is a short hand that is defined as follows:

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

8.2.3.2 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"`.

Generally means the same thing as:

1. Return NormalCompletion(`"Infinity"`).

Deleted: aggrations
Deleted: the
Deleted: type
Deleted: a
Deleted: at
Deleted: .
Deleted: F
Deleted: ,
Deleted: fields
Deleted: why
Deleted: z
Deleted: D
Deleted: D
Deleted: D
Deleted: F
Deleted: aggrations
Deleted: le
Deleted: l
Deleted: y
Deleted: the String
Deleted: s
Deleted: {[[type]]: normal, [[value]]: String
Deleted: , [[target]]:empty
Deleted: }

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

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

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

8.2.3.3 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}.

8.2.3.4 ReturnIfAbrupt

Algorithms steps that say

1. ReturnIfAbrupt(*argument*).

mean the same things as:

1. If *argument* is an abrupt completion, then return *argument*.
2. Else if *argument* is a Completion Record, then let *argument* be *argument*.[[value]].

8.2.4 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, the *referenced name* and the Boolean valued *strict reference* flag. The *base* value is either **undefined**, an Object, a Boolean, a String, a Number, or an environment record (10.2.1). A *base* value of **undefined** indicates that the Reference could not be resolved to a binding. The *referenced name* is a String or Symbol.

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

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

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

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

Deleted: argument

Deleted: <#>8.8.2 NormalCompletion
The abstract operation NormalCompletion with single *argument*, such as:
The abstraction operation NormalValue with a single *argument* is a short hand that is define as follows:
<#>If *argument* is not a Completion Record, return *argument*.
<#>If *argument* is an abrupt completion, return *argument*.
<#>Return *argument*.[[value]].
The abstraction operationabstract operation NormalCompletion with a single *argument*, su as:
<#>Return NormalCompletion(*argument*).
Is a short hand that is defined as follows:
<#>Return Completion {[[type]]: normal, [[value]]: *argument*, [[target]]:empty}.

Deleted:

Deleted: ,

Deleted: ,

Deleted: ,

Deleted: and

Deleted: lanauge

Deleted: .

Deleted: The behaviour of assignment coul instead, be explained entirely in terms of a ca analysis on the syntactic form of the left-hand operand of an assignment operator, but for o difficulty: function calls are permitted to retur references. This possibility is admitted purely the sake of host objects. No built-in ECMASc function defined by this specification returns a reference and there is no provision for a user defined function to return a reference. (Anoth reason not to use a syntactic case analysis is that it would be lengthy and awkward, affecti many parts of the specification.)

Deleted: reference

Deleted: <#>GetThisValue(V). Returns th *thisValue* component of the reference V.

8.2.4.1 GetValue (V)

1. ReturnIfAbrupt(*V*).
2. If Type(*V*) is not Reference, return *V*.
3. Let *base* be the result of calling GetBase(*V*).
4. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
5. If IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **true**, then
        i. Assert: In this case, *base* will never be **null** or **undefined**.
        ii. Let *base* be ToObject(*base*).
    b. Return the result of calling the [[Get]] internal method of *base* passing GetReferencedName(*V*) and GetThisValue(*V*) as the arguments.
6. Else *base* must be an environment record.
    a. Return the result of calling the GetBindingValue (see 10.2.1) concrete method of *base* passing GetReferencedName(*V*) and IsStrictReference(*V*) as arguments.

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

8.2.4.2 PutValue (V, W)

1. ReturnIfAbrupt(*V*).
2. ReturnIfAbrupt(*W*).
3. If Type(*V*) is not Reference, throw a **ReferenceError** exception.
4. Let *base* be the result of calling GetBase(*V*).
5. If IsUnresolvableReference(*V*), then
    a. If IsStrictReference(*V*) is **true**, then
        i. Throw **ReferenceError** exception.
    b. Let *globalObj* be the result of the abstract operation GetGlobalObject.
    c. Return the result of calling Put(*globalObj*, GetReferencedName(*V*), *W*, **false**).
6. Else if IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **true**, then
        i. Assert: In this case, *base* will never be **null** or **undefined**.
        ii. Set *base* to ToObject(*base*).
    b. Let *succeeded* be the result of calling the [[Set]] internal method of *base* passing GetReferencedName(*V*), *W*, and GetThisValue(*V*) as arguments.
    c. ReturnIfAbrupt(*succeeded*).
    d. If *succeeded* is **false** and IsStrictReference(*V*) is **true**, then throw a **TypeError** exception.
    e. Return.
7. Else *base* must be a reference whose base is an environment record. So,
    a. Return the result of calling the SetMutableBinding (10.2.1) concrete method of *base*, passing GetReferencedName(*V*), *W*, and IsStrictReference(*V*) as arguments.
8. Return.

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

8.2.4.3 GetThisValue (V)

1. ReturnIfAbrupt(*V*).
2. If Type(*V*) is not Reference, return *V*.
3. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsSuperReference(*V*), then
    a. Return the value of the *thisValue* component of the reference *V*.
5. Return GetBase(*V*).

8.2.5 The Property Descriptor Specification Type

The Property Descriptor type is used to explain the manipulation and reification of Object property attributes. Values of the Property Descriptor type are Records composed of named fields where each field's name is an

Deleted: <#>Let *V* be NormalValue(*V*).¶

Deleted: false…rue, then

Deleted: Set

Deleted: let *get* be the [[Get]] internal method of *b* otherwise let

Deleted: to

Deleted: *get* be the special [[Get]] internal method defined below

Deleted: <#>If IsSuperReference(*V*) is **false**, then ¶ <#>Return the result of calling the *get* as an internal method using of *base* as its **this** value, and passing GetReferencedName(*V*) for the argument.¶ <#>Else,¶

Deleted: *get* as an

Deleted: P

Deleted: ,…*base* must be an environment record,.

Deleted: The following [[Get]](P [, accessorThisValue internal method is used by GetValue when *V* is a prope reference with a primitive base value. It is called using *base* as is the value the internal method is called upon **this** value and with property name *P* as its argument. T following steps are taken:¶ <#>Let *O* be ToObject(*base*).¶

Deleted: method

Deleted: The only situation where such an actual

Deleted: <#>Let *V* be NormalValue(*V*).¶

Deleted: ion

Deleted: Call

Deleted: the [[…ut]] internal method of… …*lobalObj*

Deleted: false

Deleted: <#> let *put* be the [[Put]] internal method of

Deleted: *Call*

Deleted: *the*

Deleted: *put*

Deleted: P

Deleted: as an

Deleted: using …f *base* as its **this** value, and

Deleted: for the property name… *W* for the value

Deleted:)

Deleted: for the *Throw* flag

Deleted: Call

Deleted: The following [[Put]] internal method is used

Deleted: transient

Deleted: The only situations where such an actual

Deleted: <#>8.8 The List Specification Type¶

Deleted: and Property Identifier

Deleted: s

Deleted: named

Deleted: records

attribute name and its value is a corresponding attribute value as specified in 8.1.6.1. In addition, any field may be present or absent.

Property Descriptor values may be further classified as data property descriptors and accessor property descriptors based upon the existence or use of certain fields. A data property descriptor is one that includes any fields named either [[Value]] or [[Writable]]. An accessor property descriptor is one that includes any fields named either [[Get]] or [[Set]]. Any property descriptor may have fields named [[Enumerable]] and [[Configurable]]. A Property Descriptor value may not be both a data property descriptor and an accessor property descriptor; however, it may be neither. A generic property descriptor is a Property Descriptor value that is neither a data property descriptor nor an accessor property descriptor. A fully populated property descriptor is one that is either an accessor property descriptor or a data property descriptor and that has all of the fields that correspond to the property attributes defined in either 8.1.6.1 Table 5 or Table 6.

A Property Descriptor may be derived from an ECMAScript object that has properties that directly correspond to the fields of a Property Descriptor. Such a derived Property Descriptor has an additional field named [[Origin]] whose value is the object from which the Property Descriptor was derived.

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

8.2.5.1 IsAccessorDescriptor (Desc)

When the abstract operation IsAccessorDescriptor is called with property descriptor *Desc*, the following steps are taken:

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

8.2.5.2 IsDataDescriptor (Desc)

When the abstract operation IsDataDescriptor is called with property descriptor *Desc*, the following steps are taken:

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

8.2.5.3 IsGenericDescriptor (Desc)

When the abstract operation IsGenericDescriptor is called with property descriptor *Desc*, the following steps are taken:

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

8.2.5.4 FromPropertyDescriptor (Desc)

When the abstract operation FromPropertyDescriptor is called with property descriptor *Desc*, the following steps are taken:

The following algorithm assumes that *Desc* is a fully populated Property Descriptor, such as that returned from [[GetOwnProperty]] (see 8.12.1).

1. If *Desc* is **undefined**, then return **undefined**.
2. If *Desc* has an [[Origin]] field, then return *Desc*.[[Origin]].
3. Let *obj* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
4. Assert: *obj* is an extensible ordinary object with no own properties.
5. If *Desc* has a [[Value]] field, then

Deleted: For notational convenience within t specification, an object literal-like syntax can used to define a property descriptor value. Fo example, Property Descriptor {[[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true**} defi a data property descriptor. Field name order i not significant. Any fields that are not explicitl listed are considered to be absent.¶
In specification text and algorithms, dot notati may be used to refer to a specific field of a Property Descriptor. For example, if D is a property descriptor then D.[[Value]] is shortha for "the field of D named [[Value]]".¶
The Property Identifier type is used to associa a property name with a Property Descriptor. Values of the Property Identifier type are pair of the form (name, descriptor), where name is String and descriptor is a Property Descriptor value. ¶

Deleted: ion

Deleted: (15.2)

Deleted: creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name

Deleted: If IsDataDescriptor(*Desc*) is **true**

   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`value`", and Property Descriptor {[[Value]]: *Desc*.[[Value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
6. If *Desc* has a [[Writable]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`writable`", and Property Descriptor {[[Value]]: *Desc*.[[Writable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
7. If *Desc* has a [[Get]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`get`", and Property Descriptor {[[Value]]: *Desc*.[[Set]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. If *Desc* has a [[Set]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`set`", and Property Descriptor {[[Value]]: *Desc*.[[Set]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
9. If *Desc* has a [[Enumerable]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`enumerable`", and Property Descriptor {[[Value]]: *Desc*.[[Enumerable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
10. If *Desc* has a [[Configurable]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments *obj*, "`configurable`", and Property Descriptor {[[Value]]: *Desc*.[[Configurable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
11. Return *obj*.

8.2.5.5 ToPropertyDescriptor (Obj)

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

1. ReturnIfAbrupt(*Obj*).
2. If Type(*Obj*) is not Object throw a **TypeError** exception.
3. Let *desc* be the result of creating a new Property Descriptor that initially has no fields.
4. If the result of HasProperty(*Obj*, "`enumerable`") is **true**, then
   a. Let *enum* be the result of Get(*Obj*, "`enumerable`").
   b. ReturnIfAbrupt(*enum*).
   c. Set the [[Enumerable]] field of *desc* to ToBoolean(*enum*).
5. If the result of HasProperty(*Obj*, "`configurable`") is **true**, then
   a. Let *conf* be the result of Get(*Obj*, "`configurable`").
   b. ReturnIfAbrupt(*conf*).
   c. Set the [[Configurable]] field of *desc* to ToBoolean(*conf*).
6. If the result of HasProperty(*Obj*, "`value`") is **true**, then
   a. Let *value* be the result of Get(*Obj*, "`value`").
   b. ReturnIfAbrupt(*value*).
   c. Set the [[Value]] field of *desc* to *value*.
7. If the result of HasProperty(*Obj*, "`writable`") is **true**, then
   a. Let *writable* be the result of Get(*Obj*, "`writable`").
   b. ReturnIfAbrupt(*writable*).
   c. Set the [[Writable]] field of *desc* to ToBoolean(*writable*).
8. If the result of HasProperty(*Obj*, "`get`") is **true**, then
   a. Let *getter* be the result of Get(*Obj*, "`get`").
   b. ReturnIfAbrupt(*getter*).
   c. If IsCallable(*getter*) is **false** and *getter* is not **undefined**, then throw a **TypeError** exception.
   d. Set the [[Get]] field of *desc* to *getter*.
9. If the result of HasProperty(*Obj*, "`set`") is **true**, then
   a. Let *setter* be the result of Get(*Obj*, "`set`").
   b. ReturnIfAbrupt(*setter*).
   c. If IsCallable(*setter*) is **false** and *setter* is not **undefined**, then throw a **TypeError** exception.
   d. Set the [[Set]] field of *desc* to *setter*.
10. If either *desc*.[[Get]] or *desc*.[[Set]] are present, then
   a. If either *desc*.[[Value]] or *desc*.[[Writable]] are present, then throw a **TypeError** exception.
11. Set the [[Origin]] field of *desc* to *Obj*.
12. Return *desc*.

Deleted: the [[…rdinaryDefineOwnProperty]] interna
Deleted: the [[…efineOwnProperty with arguments]]
Deleted: Else, IsAccessorDescriptor(*Desc*) must be **true**
Deleted: so
Deleted: the [[…efineOwnProperty with arguments]]
Deleted: Get
Deleted: the [[…efineOwnProperty with arguments]]
Deleted: the [[…efineOwnProperty with arguments]]
Deleted: the [[…efineOwnProperty with arguments]]
Deleted: calling the [[HasProperty]] internal method of …
Deleted: calling the [[…et]] internal method of… ...b
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[…et]] internal method of… ...b
Deleted: *g*
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[…et(]] internal method of ...*bj*
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[…et]] internal method of… ...b
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[…et]] internal method of… ...b
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[…et]] internal method of… ...b

8.2.5.6 CompletePropertyDescriptor (Desc, LikeDesc)

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

1. Assert: *LikeDesc* is either a Property Descriptor or **undefined**.
2. ReturnIfAbrupt(*Desc*).
3. Assert: *Desc* is a Property Descriptor.
4. If *LikeDesc* is **undefined**, then set *LikeDesc* to Record{[[Value]]: **undefined**, [[Writable]]: **false**, [[Get]]: **undefined**, [[Set]]: **undefined**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
5. If either IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
   a. If *Desc* does not have a [[Value]] field, then set *Desc*.[[Value]] to *LikeDesc*.[[Value]].
   b. If *Desc* does not have a [[Writable]] field, then set *Desc*.[[Writable]] to *LikeDesc*.[[Writable]].
6. Else,
   a. If *Desc* does not have a [[Get]] field, then set *Desc*.[[Get]] to *LikeDesc*.[[Get]].
   b. If *Desc* does not have a [[Set]] field, then set *Desc*.[[Set]] to *LikeDesc*.[[Set]].
7. If *Desc* does not have a [[Enumerable]] field, then set *Desc*.[[Enumerable]] to *LikeDesc*.[[Enumerable]].
8. If *Desc* does not have a [[Configurable]] field, then set *Desc*.[[Configurable]] to *LikeDesc*.[[Configurable]].
9. Return *Desc*.

8.2.6 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 Clause 10.

8.3 Ordinary Object Internal Methods and Internal Data Properties

Sections 8.3-8.5 will eventually be subsectons of a new toplevel section that follow the current section 10

All ordinary objects have an internal data property called [[Prototype]]. The value of this property is either **null** or an object and is used for implementing inheritance. Data properties of the [[Prototype]] object are inherited (are visible as properties of the child object) for the purposes of get access, but not for set access. Accessor properties are inherited for both get access and set access.

Every ordinary ECMAScript object has a Boolean-valued [[Extensible]] internal data property that controls whether or not properties may be added to the object. If the value of the [[Extensible]] internal data property is **false** then additional properties may not be added to the object. In addition, if [[Extensible]] is **false** the value of [[Prototype]] internal data properties of the object may not be modified. Once the value of an object's [[Extensible]] internal data property has been set to **false** it may not be subsequently changed to **true**.

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

8.3.1 [[GetInheritance]] ( )

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

1. Return the value of the [[Prototype]] internal data property of *O*.

8.3.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of *O* is called with argument *V* the following steps are taken:

1. Assert: Either Type(*V*) is Object or Type(*V*) is Null.
2. Let *extensible* be the value of the [[Extensible]] internal data property of *O*.
3. Let *current* be the value of the [[Prototype]] internal data property of *O*.
4. If SameValue(*V*, *current*), then return **true**.
5. If *extensible* is **false**, then return **false**.

Deleted: Record

Deleted: Record

Deleted: Record

Deleted:)

Deleted: *d*

Deleted: <#>8.112.6 The Lexical Environment and Environment Record Specification Types¶
The Lexical Environment and Environment Record types are used to explain the behavio of name resolution in nested functions and blocks. These types and the operations upon them are defined in Clause 10.¶

Comment [AWB1217]: TODO

Deleted: <#>Algorithms forOrdinary Object Internal Methods and Internal Da Properties¶

Deleted: Whether or not an object can have object as its [[Prototype]] depends on the implementation. Every [[Prototype]] chain mu have finite length (that is, starting from any object, recursively accessing the [[Prototype]] internal property must eventually lead to a nu value).

Deleted: Named d

Deleted: put

Deleted: Named a

Deleted: put

Deleted: named

Deleted: named

Deleted: ¶

Deleted: native

Deleted: String

Deleted: *Throw*

Deleted: r

6. If *V* is not **null**, then
    a. Let *p* be *V*.
    b. Repeat, while *p* is not **null**
        i. If SameValue(*p*, *O*) is **true**, then return **false**.
        ii. Let *nextp* be the result of calling the [[GetInheritance]] internal method of *p* with no arguments.
        iii. ReturnIfAbrupt(*nextp*).
        iv. Let *p* be *nextp*.
7. Set the value of the [[Prototype]] internal data property of *O* to *V*.
8. Return **true**.

8.3.3 [[IsExtensible]] ( )

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

1. Return the value of the [[Extensible]] internal data property of *O*.

8.3.4 [[PreventExtensions]] ( )

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

1. Set the value of the [[Extensible]] internal data property of *O* to **false**.
2. Return **true**.

8.3.5 [[HasOwnProperty]] (P)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. If *O* does not have an own property with key *P*, return **false**
3. Return **true**.

8.3.6 [[GetOwnProperty]] (P)

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

1. Return the result of OrdinaryGetOwnProperty with arguments *O* and *P*.

8.3.6.1 OrdinaryGetOwnProperty (O, P)

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

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

Deleted: HasIntegrity

Deleted:

Deleted: Level

Deleted: HasIntegrity

Deleted: <#>If *Level* is "**nonextensible**", then ¶
<#>Return Boolean negation of the value of the [[Extensible]] internal data property of *O*¶

Deleted: ¶

Deleted: SetIntegrity

Deleted:

Deleted: Level

Deleted: SetIntegrity

Deleted: <#>Assert: Type(*B*) is Boolean.¶

Deleted: NormalCompletion(empty)

Deleted: *P* is a valid property key, either a String or a Symbol Object

Deleted: name

Deleted: *P* is a valid property key, either a String or a Symbol Object

Deleted: <#>ReturnIfAbrupt(*O*).¶

Deleted: n't

Deleted: name

Deleted: named

8.3.7 [[DefineOwnProperty]] (P, Desc)

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

1. Return the result of OrdinaryDefineOwnProperty with arguments *O*, *P*, and *Desc*.

8.3.7.1 OrdinaryDefineOwnProperty (O, P, Desc)

When the abstract operation OrdinaryDefineOwnProperty is called with Object *O*, property key *P*, and property descriptors *Desc* the following steps are taken:

1. Let *current* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
2. Let *extensible* be the value of the [[Extensible]] internal data property of *O*.
3. Return the result of ValidateAndApplyPropertyDescriptor with arguments *O*, *P*, *extensible*, *Desc*, and *current*.

8.3.7.2 IsCompatiblePropertyDescriptor (Extensible, Desc, Current)

When the abstract operation IsCompatiblePropertyDescriptor is called with Boolean value *Extensible*, and property descriptors *Desc*, and *Current* the following steps are taken:

1. Return the result of ValidateAndApplyPropertyDescriptor with arguments **undefined**, **undefined**, *Extensible*, *Desc*, and *Current*.

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

When the abstract operation ValidateAndApplyPropertyDescriptor is called with Object *O*, property key *P*, Boolean value *extensible*, and property descriptors *Desc*, and *current* the following steps are taken:

This algorithm contains steps that test various fields of the Property Descriptor *Desc* for specific values. The fields that are tested in this manner need not actually exist in *Desc*. If a field is absent then its value is considered to be **false**.

NOTE If **undefined** is passed as the *O* argument only validation is performed and no object updates are preformed.

1. Assert: If *O* is not **undefined** then *P* is a valid property key.
2. If *current* is **undefined**, then
   a. If *extensible* is **false**, then return **false**.
   b. Assert: *extensible* is **true**.
   c. If IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
      i. If *O* is not **undefined**, then create an own data property named *P* of object *O* whose [[Value]], [[Writable]], [[Enumerable]] and [[Configurable]] attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its default value.
   d. Else *Desc* must be an accessor Property Descriptor,
      i. If *O* is not **undefined**, then create an 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. Return **true**, if every field in *Desc* is absent.
4. Return **true**, if every field in *Desc* also occurs in *current* and the value of every field in *Desc* is the same value as the corresponding field in *current* when compared using the SameValue algorithm (9.12).
5. If the [[Configurable]] field of *current* is **false** then
   a. Return **false**, if the [[Configurable]] field of *Desc* is **true**.
   b. Return **false**, if the [[Enumerable]] field of *Desc* is present and the [[Enumerable]] fields of *current* and *Desc* are the Boolean negation of each other.
6. If IsGenericDescriptor(*Desc*) is **true**, then no further validation is required.
7. Else if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) have different results, then
   a. Return **false**, if the [[Configurable]] field of *current* is **false**.

Deleted: However, if *O* is a String object it ha a more elaborate [[GetOwnProperty]] internal method defined in 15.5.5.2.¶

b. If IsDataDescriptor(*current*) is **true**, then

i. If *O* is not **undefined**, then convert the property named *P* of object *O* from a data property to an accessor property. Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes and set the rest of the property's attributes to their default values.

c. Else,

i. If *O* is not **undefined**, then convert the property named *P* of object *O* from an accessor property to a data property. Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes and set the rest of the property's attributes to their default values.

8. Else if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) are both **true**, then
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Return **false**, if the [[Writable]] field of *current* is **false** and the [[Writable]] field of *Desc* is **true**.
        ii. If the [[Writable]] field of *current* is **false**, then
            1. Return **false**, if the [[Value]] field of *Desc* is present and SameValue(*Desc*.[[Value]], *current*.[[Value]]) is **false**.
    b. else the [[Configurable]] field of *current* is **true**, so any change is acceptable.
9. Else IsAccessorDescriptor(*current*) and IsAccessorDescriptor(*Desc*) are both **true**,
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Return **false**, if the [[Set]] field of *Desc* is present and SameValue(*Desc*.[[Set]], *current*.[[Set]]) is **false**.
        ii. Return **false**, if the [[Get]] field of *Desc* is present and SameValue(*Desc*.[[Get]], *current*.[[Get]]) is **false**.
10. If *O* is not **undefined**, then
    a. For each attribute field of *Desc* that is present, set the correspondingly named attribute of the property named *P* of object *O* to the value of the field.
11. Return **true**.

NOTE Step 8.b allows any field of Desc to be different from the corresponding field of current if current's [[Configurable]] field is **true**. This even permits changing the [[Value]] of a property whose [[Writable]] attribute is **false**. This is allowed because a **true** [[Configurable]] attribute would permit an equivalent sequence of calls where [[Writable]] is first set to **true**, a new [[Value]] is set, and then [[Writable]] is set to **false**.

8.3.8 [[HasProperty]](P)

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

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

8.3.9 [[Get]] (P, Receiver)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
3. ReturnIfAbrupt(*desc*).
4. If *desc* is **undefined**, then
    a. Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.
    b. ReturnIfAbrupt(*parent*).
    c. If *parent* is **null**, then return **undefined.**
    d. Return the result of calling the [[Get]] internal method of *parent* with arguments *P* and *Receiver*.

Deleted: However, if *O* has an [[BuiltinBrand]] internal data property whose value is BuiltinArray *O* also has a more elaborate [[DefineOwnProperty]] internal method defined in 15.4.5.1.¶

Deleted: 10

Deleted: ~~8.3.11 [[DefaultValue]] (hint)~~¶
~~When the [[DefaultValue]] internal method of *O* is called with hint String, the following steps are taken:~~¶
~~<#>Let *toString* be the result of Get(*O*, "toString").~~¶
~~<#>ReturnIfAbrupt(*toString*).~~¶
~~<#>If IsCallable(*toString*) is **true** then,~~¶
~~<#>Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as *thisArgument* and an empty List as *argumentsList*.~~¶
~~<#>ReturnIfAbrupt(*str*).~~¶
~~<#>If *str* is a primitive value, return *str*.~~¶
~~<#>Let *valueOf* be the result of Get(*O*, "valueOf").~~¶
~~<#>ReturnIfAbrupt(*valueOf*).~~¶
~~<#>If IsCallable(*valueOf*) is **true** then,~~ ¶
~~<#>Let *val* be the result of calling the [[Call]] internal method of *valueOf*, with *O* as *thisArgument* and an empty argument list.~~¶
~~<#>ReturnIfAbrupt(*val*).~~¶
~~<#>If *val* is a primitive value, return *val*.~~¶
~~<#>Throw a **TypeError** exception.~~¶
~~When the [[DefaultValue]] internal method of *O* is called with hint Number, the following steps are taken:~~¶
~~<#>Let *valueOf* be the result of Get(*O*, "valueOf").~~¶
~~<#>ReturnIfAbrupt(*valueOf*).~~¶
~~<#>If IsCallable(*valueOf*) is **true** then,~~¶
~~<#>Let *val* be the result of calling the [[Call]] internal method of *valueOf*, with *O* as *thisArgument* and an empty List as *argumentsList*.~~¶
~~<#>ReturnIfAbrupt(*val*).~~¶
~~<#>If *val* is a primitive value, return *val*.~~¶

Deleted: Get

Deleted: *desc*

Deleted: *desc*

Deleted: *desc*

Deleted: **undefined**

Deleted: **false**

Deleted: P

Deleted: P

Deleted: name

Deleted: P

Deleted: ,

Deleted: *P* is a valid property key, either a String or a Symbol Object

Deleted: the [[

Deleted: OrdinaryGetOwnProperty

Deleted:]] internal method of

Deleted: s

Deleted: *O* with property name and

Deleted: return **undefined**.

Deleted: P

Deleted: s

5. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
6. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].
7. If *getter* is **undefined**, return **undefined**.
8. Return the result of calling the [[Call]] internal method of *getter* with *Receiver* as the *thisArgument* and an empty List as *argumentsList*.

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

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

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

8.3.11 [[Invoke]] (P, ArgumentsList, Receiver)

When the [[Invoke]] internal method of *O* is called with property key *P*, List *ArgumentsList*, and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Assert: *argumentsList* is a List.
3. Let *method* be the result of calling the [[Get]] internal method of *O* with arguments *P*, and *Receiver*.
4. ReturnIfAbrupt(*method*).
5. If Type(*method*) is not Object, throw a **TypeError** exception.
6. If IsCallable(*method*) is **false**, throw a **TypeError** exception.
7. Return the result of calling the [[Call]] internal method of *method* with *Receiver* as the *thisArgument* and *ArgumentsList* as *argumentsList*.

8.3.12 [[Delete]] (P)

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

Deleted: *targetThis*
Deleted: Put]
Deleted: P
Deleted: Throw
Deleted: Put
Deleted: P
Deleted: and Boolean flag *Throw*
Deleted: *P* is a valid property key, either a Str or a Symbol Object
Deleted: <#>If the result of calling the [[CanPut]] internal method of *O* with argument **false**, then¶
<#>If *Throw* is **true**, then throw a **TypeErr** exception.¶
<#>Else return **undefined**.¶
Deleted: Ordinary[[GetOwnProperty
Deleted:]] internal method of
Deleted: s…*O* with argumentand
Deleted: *desc*
Deleted: P
Deleted: s
Deleted: <#>If Type(*Receiver*) is not Object, return **false**.¶
<#>Let *newDesc* be the Property Descriptor {[[Value]]: *V*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
Deleted: be
Deleted: SameValue(*O*, *Receiver*)…is not
Deleted: the [[
Deleted: Ordinary
Deleted: *]] internal method of*
Deleted: *OO* passing , …,
Deleted: ,…*valueDesc*, and *Throw* as
Deleted: *O* and …*eceiver* are different valu
Deleted: <#>If Type(*Receiver*) is not
Deleted: <#>Let *desc* be the result of calling
Deleted: *desc*
Deleted: *desc*
Deleted: which cannot be **undefined**
Deleted: <#>If *accessorThisValue* is not
Deleted: Else create a named data property
Deleted: ~~Let *newDesc* be the Property~~
Deleted: ~~8.3.9 [[HasProperty]] (P)~~¶
Deleted: <#>8.12.5 [[Put]] (P, V, Thro
Deleted: <#>Return.¶
Deleted: <#>8.12.6 [[HasProperty]] (P
Deleted: , Throw
Deleted: name …ey *P* and the Boolean flag

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

8.3.13 [[Enumerate]] ()

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

1. Return an Iterator object (reference xxxx) whose next method iterates over all the String valued keys of enumerable property keys of *O*. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.

Enumerated properties do not include properties whose property key is a Symbol. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name. The values of [[Enumerable]] attributes are not considered when determining if a property of a prototype object is shadowed by a previous object on the prototype chain.

The following is an informative algorithm that conforms to these rules

1. Let *obj* be *O*.
2. Let *proto* be the result of calling the [[GetInheritance]] internal method of *O* with no arguments.
3. ReturnIfAbrupt(*proto*).
4. If *proto* is the value **null**, then
   a. Let *propList* be a new empty List.
5. Else
   a. Let *propList* be the result of calling the [[Enumerate]] internal method of *proto*.
6. ReturnIfAbrupt(*propList*).
7. For each *name* that is the property key of an own property of *O*
   a. If Type(*name*) is String, then
      i. Let *desc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *name*.
      ii. If *name* is an element of *propList*, then remove *name* as an element of *propList*.
      iii. If *desc*.[[Enumerable]] is **true**, then add *name* as an element of *propList*.
8. Order the elements of *propList* in an implementation defined order.
9. Return *propList*.

8.3.14 [[OwnPropertyKeys]] ( )

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

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

8.3.15 ObjectCreate(proto, internalDataList) Abstract Operation

The abstract operation ObjectCreate with argument *proto* (an object or null) is used to specify the runtime creation of new ordinary objects. The optional argument *internalDataList* is a List of the names of internal data property names that should be defined as part of the object. If the list is not provided, an empty List is used. If

Deleted: *P* is a valid property key, either a String or a Symbol Object

Deleted: <#>ReturnIfAbrupt(*O*).¶

Deleted: the [[

Deleted: OrdinaryGetOwnProperty

Deleted:]] internal method of

Deleted: s

Deleted: *O* with property nameand

Deleted: <#>Else if *Throw*, then throw a **TypeError** exception.¶

Comment [AWB618]: TODO

Comment [AWB619]: TODO: Finish this up, and tur into iterator definition include a next method

Deleted: <#>8.3.910 [[DefineOwnProperty]] (P, Desc, Throw)¶
When the [[DefineOwnProperty]] internal method of *O* i called with property key *P* and property descriptor *Desc* the following steps are taken:¶
<#>Return the result of OrdinaryDefineOwnProperty with arguments *O*, *P*, and *Desc*.¶
<#>8.3.10.1 OrdinaryDefineOwnProperty (O, P, Desc)¶
When the abstract operation OrdinaryDefineOwnPropert called with Object *O*, property key *P*, and property descriptors *Desc* the following steps are taken:¶
<#>Let *current* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.¶
<#>Let *extensible* be the value of the [[Extensible]] intern data propertry of *O*.¶
<#>Return the result of ValidateAndApplyPropertyDescriptor with arguments *O*, *I* *extensible*, *Desc*, and *current*.¶
<#>8.3.10.2 IsCompatableDescriptor (Extensibl Desc, Current)¶
When the abstract operation IsCompatablePropertyDescri is called with Boolean value *Extensible*, and property descriptors *Desc*, and *Current* the following steps are taken:¶
<#>Return the result of ValidateAndApplyPropertyDescriptor with arguments **undefined**, **undefined**, *Extensible*, *Desc*, and *Current*.¶

Comment [AWB620]: TODO

Comment [AWB621]: TODO: Finish this up, and tur into iterator definition include a next method.

Deleted: <#>8.123.112 [[IterateKeys]] ( )¶

Deleted: *result*

Deleted: <#>If Type(*O*) is not Object, return *result*.¶

Deleted: <#>If *P* is not a private Symbol, then¶

Deleted: *result*

Comment [AWB1322]: TODO: need to define, retur an iterator object over the elements of an internal list.

Deleted: *result*

Deleted: *list*

Deleted: <#>8.3.14 [[Freeze]] ( )¶

Deleted: optional

no arguments are provided %ObjectPrototype% is used as its value. This abstract operation performs the following steps:

1. If *proto* was not provided, let *proto* be the intrinsic object %ObjectPrototype%.
2. Let *obj* be a newly created ECMAScript object with an internal data property for each name in *internalDataList*.
3. Set *obj*'s essential internal methods to the default ordinary object definitions specified in 8.3.
4. Set the [[Prototype]] internal data property of *obj* to *proto*.
5. Set the [[Extensible]] internal data property of *obj* to **true**.
6. Return *obj*.

8.3.16 Ordinary Function Objects

Ordinary function objects encapsulate parameterised ECMAScript code closed over a lexical environment and support the dynamic evaluation of that code. An ordinary function object is an ordinary object and has the same internal data properties and (except as noted below) the same internal methods as other ordinary objects.

Ordinary function objects have the additional internal data properties listed in Table 13.

Ordinary function objects provide alternative definitions for the [[Get]] and [[GetOwnProperty]] internal methods. These alternatives prevent the value of strict mode function from being revealed as the value of a function object property named "`caller`". These alternative definitions exist sole to preclude a non-standard legacy feature of some ECMAScript implementations from revealing information about strict mode callers. If an implementation does not provide such a feature, it need not implement these alternative internal methods for ordinary function objects.

Deleted: It

Deleted: <#>8.3.195 Ordinary Functio Objects¶

Deleted: z

Deleted: They also have a [[BuiltinBrand]] internal data property whose value is **BuiltinFunction**.

Deleted: P

Table 13 -- Internal Data Properties of Ordinary Function Objects

Internal Data Property	*Type*	*Description*
[[Scope]]	Lexical Environment	The Lexical Environment that the function was closed over. Is used as the outer environment when evaluating the code of the function.
[[FormalParameters]]	Parse Node	The root parse node of the source code that defines the function's formal parameter list.
[[FunctionKind]]	String	Either "`normal`" or "`generator`".
[[Code]]	Parse Node	The root parse node of the source code that defines the function's body.
[[Realm]]	Realm Record	The Code Realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.
[[ThisMode]]	(lexical, strict, global)	Defines how `this` references are interpreted within the formal parameters and code body of the function. **lexical** means that `this` refers to the **this** value of a lexically enclosing function. **strict** means that the **this** value is used exactly as provided by an invocation of the function. **global** means that a this value of **undefined** is interpreted as a reference to the global object.
[[Strict]]	Boolean	**true** if this is a strict mode function, **false** this is not a strict mode function.
[[Home]]	Object	If the function uses `super`, this is the object whose [[GetInheritance]] provides the object where super property lookups begin. Not present for functions that don't reference `super`.
[[MethodName]]	String or Symbol	If the function uses `super`, this is the property keys that is used for unqualified references to `super`. Not present for functions that don't reference `super`.

Ordinary function objects all have the [[Call]], [[Get]] and [[GetOwnProperty]] internal methods defined here. Ordinary functions that are also constructors in addition have the [[Construct]] internal method.

Deleted: P

Deleted: i

8.3.16.1 [[Call]] (thisArgument, argumentsList)

Deleted: Internal Method

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

1. If *F*'s [[Code]] internal data property has the vaue **undefined**, then throw a **TypeError** exception.
2. Let *callerContext* be the running execution context.
3. If, *callerContext* is not already suspended, then Suspend *callerContext*.
4. Let *calleeContext* be a new ECMAScript Code execution context.
5. Let *calleeRealm* be the value of *F's* [[Realm]] internal data property.
6. Set *calleeContext's* Realm to *calleeRealm*.
7. Let *thisMode* be the value of *F*'s [[ThisMode]] internal data property.
8. If *thisMode* is lexical, then
    a. Let *localEnv* be the result of calling NewDeclarativeEnvironment passing the value of the [[Scope]] internal data property of *F* as the argument.
9. Else,
    a. If *thisMode* is strict, set *thisValue* to *thisArgument*.
    b. Else
        i. if *thisArgument* is **null** or **undefined**, then
            1. Set *thisValue* to *calleeRealm*.[[globalThis]].
        ii. Else if Type(*thisArgument*) is not Object, set the *thisValue* to ToObject(*thisArgument*).
        iii. Else set the *thisValue* to *thisArgument*.
    c. Let *localEnv* be the result of calling NewFunctionEnvironment passing *F* and *thisValue* as the arguments.

10. Set the LexicalEnvironment of *calleeContext* to *localEnv*.
11. Set the VariableEnvironment of *calleeContext* to *localEnv*.
12. Push *calleeContext* onto the execution context stack; *calleeContext* is now the running execution context.
13. Let *status* be the result of performing Function Declaration Instantiation using the function *F*, *argumentsList* , and *localEnv* as described in 10.5.3.
14. If *status* is an abrupt completion, then
    a. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
    b. Return *status*.
15. Let *result* be the result of EvaluateBody of the production that is the value of *F*'s [[Code]] internal data property passing *F* as the argument.
16. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
17. Return *result*.

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

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

8.3.16.2 [[Construct]] (argumentsList)

The [[Construct]] internal method for an ordinary Function object *F* is called with a single parameter *argumentsList* which is a possibly empty List of ECMAScript language values. The following steps are taken:

1. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

8.3.16.2.1 OrdinaryConstruct (F, argumentsList)

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

1. Let *creator* be the result of Get(*F*, @@create).
2. ReturnIfAbrupt(*creator*).
3. If *creator* is not **undefined**, then
    a. If IsCallable(*creator*) is **false**, then throw a **TypeError** exception.
    b. Let *obj* be the result of calling the [[Call]] internal method of *creator* with arguments *F* and an empty List.
4. Else *creator* is **undefined** so fall back to object creation defaults
    a. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%ObjectPrototype%"`).
5. ReturnIfAbrupt(*obj*).
6. If Type(*obj*) is not Object, then throw a **TypeError** exception.
7. Let *result* be the result of calling the [[Call]] internal method of *F*, providing *obj* and *argumentsList* as the arguments.
8. ReturnIfAbrupt(*result*).
9. If Type(*result*) is Object then return *result*.
10. Return *obj*.

8.3.16.3 [[Get]] (P, Receiver)

When the [[Get]] internal method of ordinary function object *F* is called with property key *P* and ECMAScript language value *Receiver* the following steps are taken:

1. Let *v* be the result of calling the default ordinary object [[Get]] internal method (8.3.7) on *F* passing *P* and *Receiver* as arguments.
2. ReturnIfAbrupt(*v*).
3. If *P* is `"caller"` and *v* is a strict mode Function object, return **null**.
4. Return *v*.

If an implementation does not provide such a built-in `caller` method for `Function.prototype` then it must not use this definition. Instead the ordinary object [[Get]] internal method is used.

Deleted:

Comment [AWB 323]: May need to update section number

Deleted: evaluating

Deleted: FunctionBody

Deleted: <#>If *result*.type is return then retu NormalCompletion(*result*.[[value]]).¶

Deleted: Internal Method

Deleted: with arguments

Deleted: and

Deleted: , fallBackProto

Comment [AWB1424]: At Jan 29, 2012 TC serveral peopled suggest that this fall back w unnecessary complexity and that it should thi throw. However, that means that a ordinary function whose __proto__ is set to undefined will throw if newed. I'm not sure that is desirable. It's a breaking change for the realit web.

Deleted: <#>Let *proto* be the result of Get `"prototype"`).¶
<#>ReturnIfAbrupt(*proto*).¶
<#>If Type(*proto*) is Object, let *obj* be the result of the abstract operation ObjectCreate with argument *proto*.¶
<#>Else, let *obj* be the result of the abstract operation ObjectCreate.¶

Deleted: NormalCompletion(

Deleted:)

Deleted: P

Deleted: P

Deleted: P

Deleted: undefined

8.3.16.4 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of ordinary function object *F* is called with property key *P*, the following steps are taken:

1. Let *v* be the result of calling the default ordinary object [[GetOwnProperty]] internal method (8.3.6) on *F* passing *P* as the argument.
2. ReturnIfAbrupt(*v*).
3. If IsDataDescriptor(*v*) is **true**, then
   a. If *P* is `"caller"` and *v*.[[Value]] is a strict mode Function object, then
      i. Set *v*.[[Value]] to **null**.
4. Return *v*.

If an implementation does not provide such a built-in `caller` method for `Function.prototype` then it must not use this definition. Instead the ordinary object [[GetOwnProperty]] internal method is used.

8.3.16.5 FunctionAllocate Abstract Operation

The abstract operation FunctionAllocate requires the one argument, *functionPrototype* and accepts one optional argument, *functionKind*. FunctionAllocate performs the following steps:

1. Assert: Type(*functionPrototype*) is Object.
2. Assert: If *functionKind* is present, its value is either `"normal"` or `"generator"`.
3. If *functionKind* is not present, then let *functionKind* be `"normal"`.
4. Let *F* be a newly created ordinary function object with the internal data properties listed in Table 13. All of those internal data properties are initialized to **undefined**.
5. Set *F*'s essential internal methods except for [[Get]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.
6. Set *F*'s essential internal methods for [[Call]], [[Get]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.16.
7. Set the [[FunctionKind]] internal data property of *F* to *functionKind*.
8. Set the [[Prototype]] internal data property of *F* to *functionPrototype*.
9. Set the [[Extensible]] internal data property of *F* to **true**.
10. Set the [[Realm]] internal data property of *F* to the running execution context's Realm.
11. Return *F*.

8.3.16.6 FunctionInitialise Abstract Operation

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

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

Deleted: undefined
Deleted: *v*.[[Value]]
Deleted: s
Deleted:
Deleted:
Deleted: 8.3
Deleted: 8.3
Deleted:
Deleted: 8.3.16
Deleted: 8.3.15
Deleted: ize
Deleted: ize
Deleted: ize
Moved (insertion) [5]
Moved (insertion) [4]
Deleted: Call the [[
Deleted: Own
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted:
Deleted: and
Deleted: Perform
Deleted: <#>Set the [[Prototype]] internal data property *F* to *functionPrototype*.¶
Moved up [5]: <#>Let *len* be the ExpectedArgumentCount of *ParameterList*.¶
Deleted: <#>Let *len* be the ExpectedArgumentCount of ExpectedArgumentCount of *ParameterList*.¶
Moved up [4]: <#>Call the [[DefineOwnProperty]] internal method of *F* with arguments `"length"` and Property Descriptor {[[Value]]: *len*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}¶
<#>If *Strict* is **true**, then¶
<#>Perform the AddRestrictedFunctionProperties abstract operation with argument *F*. ¶

8.3.16.7 FunctionCreate Abstract Operation

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

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

Deleted: ize

8.3.16.8 GeneratorFunctionCreate Abstract Operation

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

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

Deleted: ize

8.3.16.9 AddRestrictedFunctionProperties Abstract Operation

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

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

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

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

Deleted: Call the
Deleted: [[
Deleted: Own
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: Call
Deleted: [[
Deleted: Own
Deleted:]] internal method of
Deleted: with arguments
Deleted: and
Deleted: Let *F* be
Deleted: Create
Deleted: SetIntegrity
Deleted: with argument "**`nonextensible`**"
Deleted: ¶
Let %ThrowTypeError% be *F*.

8.3.16.10 MakeConstructor Abstract Operation

The abstract operation MakeConstructor requires a Function argument *F* and optionally, a Boolean *writablePrototype* and an object *prototype*. If *prototype* is provided it is assume to already contain, if needed, a **`"constructor"`** property whose value is *F*. This operation converts *F* into a constructor by performs the following steps:

1. Let *installNeeded* be **false**.
2. If the *prototype* argument was not provided,then

a. Let *installNeeded* be **true**.
b. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
3. If the *writablePrototype* argument was not provided,then
   a. Let *writablePrototype* be **true**.
4. Set *F*'s essential internal method [[Construct]] to the definition specified in 8.3.15.2.
5. If *installNeeded*, then
   a. Call the [[DefineOwnProperty]] internal method of *prototype* with arguments **`"constructor"`** and Property Descriptor {[[Value]]: *F*, [[Writable]]: *writablePrototype*, [[Enumerable]]: **false**, [[Configurable]]: *writablePrototype* }
7. Call the [[DefineOwnProperty]] internal method of *F* with arguments **`"prototype"`** and Property Descriptor {[[Value]]: *prototype* , [[Writable]]: *writablePrototype* , [[Enumerable]]: **false**, [[Configurable]]: **false**}.
8. Return.

8.4 Built-in Exotic Object Internal Methods and Data Fields

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

8.4.1 Bound Function Exotic Objects

A *bound function* is an exotic object that wrappers another function object. A bound function is callable (it has [[Call]] and [[Construct]] internal methods). Calling a bound function generally results in a call of its wrappered function.

Bound function objects do not have the internal data properties of ordinary function objects defined in Table 13. Instead they have the internal data properties defined in Table 14.

Table 14 -- Internal Data Properties of Exotic Bound Function Objects

Internal Data Property	*Type*	*Description*
[[BoundTargetFunction]]	Callable Object	The wrappered function object.
[[BoundThis]]	Any	The value that is always passed as the **this** value when calling the wrappered function.
[[BoundArguments]]	List of Any	A list of values that whose elements are used as the first arguments to any call to the wrappered function.

Unlike ordinary function objects, bound function objects do not use alternative definitions of the [[Get]] and [[GetOwnPropety]] internal methods. Bound function objects provide all of the essential internal methods as specified in 8.3. However, they use the following definitions for the essential internal methods of function objects.

8.4.1.1 [[Call]]

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

1. Let *boundArgs* be the value of *F's* [[BoundArguments]] internal data property.
2. Let *boundThis* be the value of *F's* [[BoundThis]] internal data property.
3. Let *target* be the value of *F's* [[BoundTargetFunction]] internal data property.
4. Let *args* be a new list containing the same values as the list *boundArgs* in the same order followed by the same values as the list *argumentsList* in the same order.
5. Return the result of calling the [[Call]] internal method of *target* providing *boundThis* as *thisArgument* and providing *args* as *argumentsList*.

Deleted: t

Deleted: at

Deleted: They also have a [[BuiltinBrand]] internal dat property whose value is **BuiltinFunction**.

Deleted: P

8.4.1.2 [[Construct]]

When the [[Construct]] internal method of an exotic bound function object, *F* that was created using the bind function is called with a list of arguments *ExtraArgs*, the following steps are taken:

1. Let *target* be the value of *F's* [[BoundTargetFunction]] internal data property.
2. If *target* has no [[Construct]] internal method, a **TypeError** exception is thrown.
3. Let *boundArgs* be the value of *F's* [[BoundArguments]] internal data property.
4. Let *args* be a new list containing the same values as the list *boundArgs* in the same order followed by the same values as the list *ExtraArgs* in the same order.
5. Return the result of calling the [[Construct]] internal method of *target* providing *args* as the arguments.

8.4.1.3 BoundFunctionCreate Abstract Operation

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

1. Let *proto* be the intrinsic %FunctionPrototype%.
2. Let *obj* be a newly created ECMAScript object.
3. Set *obj*'s essential internal methods to the default ordinary object definitions specified in 8.3.
4. Set the [[Call]] internal method of *obj* as described in 8.4.1.1.
5. If *targetFunction* has a [[Construct]] internal method, then
   a. Set the [[Construct]] internal method of *obj* as described in 8.4.1.2.
6. Set the [[Prototype]] internal data property of *obj* to *proto*.
7. Set the [[Extensible]] internal data property of *obj* to **true**.
8. Set the [[BoundTargetFunction]] internal data property of *obj* to *targetFunction*.
9. Set the [[BoundThis]] internal data property of *obj* to the value of *boundThis*.
10. Set the [[BoundArguments]] internal data property of *obj* to *boundArgs*.
11. Return *obj*.

8.4.2 Array Exotic Objects

An *Array object* is an exotic object that gives special treatment to a certain class of property names. A property name *P* (in the form of a String value) is an *array index* if and only if ToString(ToUint32(*P*)) is equal to *P* and ToUint32(*P*) is not equal to $2^{32}-1$. A property whose property name is an array index is also called an *element*. Every Array object has a `length` property whose value is always a nonnegative integer less than 2^{32}. The value of the `length` property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the `length` property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the `length` property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to own properties of an Array object and is unaffected by `length` or array index properties that may be inherited from its prototypes.

Exotic Array objects have the same internal data properties as ordinary objects. They also have an [[ArrayInitialisationState]] internal data property.

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

Exotic Array objects provide an alternative definition for the [[DefineOwnProperty]] internal methods. Except for that internal methods, exotic Array objects provide all of the other essential internal methods as specified in 8.3.

8.4.2.1 [[DefineOwnProperty]] (P, Desc)

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

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

Deleted: Object

Deleted: *F*

Deleted: <#>Add the [[BuiltinBrand]] interna data property with value BuiltinFunction to *obj*.

Deleted: Array objects

Deleted: z

Deleted: Exotic Array objects have the same internal data properties as ordinary objects. They also have a [[BuiltinBrand]] internal data property whose value is **BuiltinArray**.¶

Deleted: s

Deleted: [[SetP]] and

Deleted: these two

Deleted: <#>8.4.2.1 [[SetP]] (P, V, Receiver)¶
When the [[SetP]] internal method of an exotic Array object *O* is called with property k *P*, value *V*, and ECMAScipt language value *Receiver*, the following steps are taken:¶
<#>Assert: *P* is a valid property key, either a String or a Symbol Object.¶
<#>Let *ownDesc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and
<#>ReturnIfAbrupt(*ownDesc*).¶
<#>If *desc* is **undefined**, then¶
<#>Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.¶
<#>ReturnIfAbrupt(*parent*).¶
<#>If *parent* is not **null**, then ¶
<#>Return the result of calling the [[SetP internal methods of *parent* with arguments *V*, and *Receiver*.¶
<#>Else,¶
<#>If Type(*Receiver*) is not Object, retur **false**.¶
<#>Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).¶
<#>If IsDataDescriptor(*ownDesc*) is **true**, then
<#>If *ownDesc*.[[Writable]] is **false**, return **false**.¶
<#>If SameValue(*O*, *Receiver*) is **true**, then
<#>Let *valueDesc* be the Property Descriptor {[[Value]]: *V*}.¶
<#>If *P* is `"length"`, then¶
<#>Return the result of calling ArraySetLength with arguments *O*, and *valueDesc*.¶
<#>Else, ¶
<#>Return the result of calling OrdinaryDefineOwnProperty with arguments *O*, *P*, and *valueDesc*.¶
<#>Else *O* and *Receiver* are different values
<#>If Type(*Receiver*) is not Object, retur **false**.¶
<#>Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).¶
<#>If IsAccessorDescriptor(*desc*) is **true**, then
<#>Let *setter* be *desc*.[[Set]].¶
<#>If *setter* is **undefined**, return **false**.¶

Deleted: *P* is a valid property key, either a Str or a Symbol Objec

Deleted: t

2. If *P* is **"length"**, then
    a. Return the result of calling ArraySetLength with arguments *A*, and *Desc*.
3. Else if *P* is an array index, then
    a. Let *oldLenDesc* be the result of calling the [[GetOwnProperty]] internal method of *A* passing **"length"** as the argument. The result will never be **undefined** or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.
    b. Let *oldLen* be *oldLenDesc*.[[Value]].
    c. Let *index* be ToUint32(*P*).
    d. Assert: *index* will never be an abrupt completion.
    e. If *index* ≥ *oldLen* and *oldLenDesc*.[[Writable]] is **false**, then return **false**.
    f. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, *P*, and *Desc* as arguments.
    g. ReturnIfAbrupt(*succeeded*).
    h. If *succeeded* is **false**, then return **false**.
    i. If *index* ≥ *oldLen*
        i. Set *oldLenDesc*.[[Value]] to *index* + 1.
        ii. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *oldLenDesc* as arguments.
        iii. ReturnIfAbrupt(*succeeded*).
    j. Return **true**.
4. Return the result of calling OrdinaryDefineOwnProperty passing *A*, *P*, and *Desc* as arguments.

8.4.2.2 ArrayCreate Abstract Operation

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

1. If the *proto* argument was not passed, then let *proto* be the intrinsic object %ArrayPrototype%.
2. Let *A* be a newly created Array exotic object.
3. Set *A*'s essential internal methods to the default ordinary object definitions specified in 8.3.
4. Set the [[DefineOwnProperty]] internal method of *A* as specified in 8.4.2.1.
5. Set the [[Prototype]] internal data property of *A* to *proto*.
6. Set the [[Extensible]] internal data property of *A* to **true**.
7. If *length* is not **undefined**, then
    a. Set the [[ArrayInitialisationState]] internal data property of *A* to **true**.
8. Else
    a. Set the [[ArrayInitialisationState]] internal data property of *A* to **false**.
    b. Let *length* be 0.
9. Call OrdinaryDefineOwnProperty with arguments *A*, **"length"** and Property Descriptor {[[Value]]: *length*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
10. Return *A*.

8.4.2.3 ArraySetLength Abstract Operation

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

1. If the [[Value]] field of *Desc* is absent, then
    a. Return the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *Desc* as arguments.
2. Let *newLenDesc* be a copy of *Desc*.
3. Let *newLen* be ToUint32(*Desc*.[[Value]]).
4. If *newLen* is not equal to ToNumber(*Desc*.[[Value]]), throw a **RangeError** exception.
5. Set *newLenDesc*.[[Value]] to *newLen*.
6. Let *oldLenDesc* be the result of calling the [[GetOwnProperty]] internal method of *A* passing **"length"** as the argument. The result will never be **undefined** or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.
7. Let *oldLen* be *oldLenDesc*.[[Value]].
8. If *newLen* ≥ *oldLen*, then

Deleted: ReturnIfAbrupt(*index*)

Deleted: Reject i

Deleted: ECMAScript

Deleted: <#>Set the [[SetP]] internal method of *A* as specified in 8.4.2.1.¶

Deleted: 2

Deleted: the intrinsic object %ArrayPrototype%

Deleted: <#>Set the [[BuiltinBrand]] internal data prop of *A* to the value BuiltinArray.¶

Deleted: z

Deleted: z

Deleted: *O*

Deleted: <#>Let *oldLenDesc* be the result of calling the [[GetOwnProperty]] internal method of *A* passing **"lengt** as the argument. The result will never be **undefined** or an accessor descriptor because Array objects are created with length data property that cannot be deleted or reconfigured. <#>Let *oldLen* be *oldLenDesc*.[[Value]].¶

Comment [AWB1425]: Note that if [[Value]] is an object this sequence will call its valueOf method twice That seems undesirable, but it is the legacy behaviour going back to at least ES3

Comment [AWB1426]: See bug https://bugs.ecmascript.org/show_bug.cgi?id=1200 for why these two lines moved.

a. Return the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.

9. If *oldLenDesc*.[[Writable]] is **false**, then return **false**.
10. If *newLenDesc*.[[Writable]] is absent or has the value **true**, let *newWritable* be **true**.
11. 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**.
12. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.
13. ReturnIfAbrupt(*succeeded*).
14. If *succeeded* is **false**, return **false**.
15. While *newLen* < *oldLen* repeat,
    a. Set *oldLen* to *oldLen* – 1.
    b. Let *deleteSucceeded* be the result of calling the [[Delete]] internal method of *A* passing ToString(*oldLen*).
    c. ReturnIfAbrupt(*succeeded*).
    d. If *deleteSucceeded* is **false**, then
        i. Set *newLenDesc*.[[Value]] to *oldLen+1*.
        ii. If *newWritable* is **false**, set *newLenDesc*.[[Writable]] to **false**.
        iii. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.
        iv. ReturnIfAbrupt(*succeeded*).
        v. Return **false**.
16. If *newWritable* is **false**, then
    a. Call OrdinaryDefineOwnProperty passing *A*, **"length"**, and Property Descriptor{[[Writable]]: **false**} as arguments. This call will always return **true**.
17. Return **true**.

8.4.3 String Exotic Objects

A *String object* is an exotic object that encapsulates a String value and exposes virtual array index data properties corresponding to the individual code unit elements of the string value. Exotic String objects always have a data property named **"length"** whose value is the number of code unit elements in the encapsulated String value. Both the code unit data properties and the **"length"** property are non-writable and non-configurable.

Exotic String objects have the same internal data properties as ordinary objects. They also have a [[StringData]] internal data property.

Exotic String objects provide alternative definitions for the following internal methods. All of the other exotic String object essential internal methods that are not defined below are as specified in 8.3.

8.4.3.1 [[HasOwnProperty]] (P)

When the [[HasOwnProperty]] internal method of exotic String object *O* is called with property key *P*, the following steps are taken:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *has* be the result of calling the ordinary object [[HasOwnProperty]] internal method (8.3.5) on *O* with argument *P*.
3. ReturnIfAbrupt(*has*).
4. If *has* is **true**, then return **true**.
5. If Type(*P*) is not String, then return **false**.
6. Let *index* be ToInteger(*P*).
7. Assert: *index* is not an abrupt completion.
8. Let *absIntIndex* be ToString(abs(*index*)).
9. If SameValue(*absIntIndex*, *P*) is **false** return **false**.

Deleted: and a [[BuiltinBrand]] internal data property whose value is **BuiltinStringWrapp**

Deleted: Assert: *P* is a valid property key, eith a String or a Symbol Object.

Deleted: ReturnIfAbrupt(*index*).

Deleted: <#>ReturnIfAbrupt(*absIntIndex*).¶

10. Let *str* be the String value of the [[StringData]] internal property of *O*, if the value of [[StringData]] is undefined the empty string is used as its value.
11. Let *len* be the number of elements in *str*.
12. If *len* ≤ *index*, return **false**.
13. Return **true**.

8.4.3.2 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of an exotic String object *S* is called with property key *P* the following steps are taken:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *desc* be the result of OrdinaryGetOwnProperty(*S*, *P*).
3. ReturnIfAbrupt(*desc*).
4. If *desc* is not **undefined** return *desc*.
5. If Type(*P*) is not String, then return **undefined**.
6. Let *index* be ToInteger(*P*).
7. Assert: *index* is not an abrupt completion.
8. Let *absIntIndex* be ToString(abs(*index*)).
9. If SameValue(*absIntIndex*, *P*) is **false** return **undefined**.
10. Let *str* be the String value of the [[StringData]] internal data property of *S*, if the value of [[StringData]] is undefined the empty string is used as its value.
11. Let *len* be the number of elements in *str*.
12. If *len* ≤ *index*, return **undefined**.
13. Let *resultStr* be a String value of length 1, containing one code unit from *str*, specifically the code unit at position *index*, where the first (leftmost) element in *str* is considered to be at position 0, the next one at position 1, and so on.
14. Return a Property Descriptor { [[Value]]: *resultStr*, [[Enumerable]]: **true**, [[Writable]]: **false**, [[Configurable]]: **false** }.

8.4.3.3 [[DefineOwnProperty]] (P, Desc)

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

1. Let *current* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
2. Let *extensible* be the value of the [[Extensible]] internal data property of *O*.
3. Return the result of ValidateAndApplyPropertyDescriptor with arguments *O*, *P*, *extensible*, *Desc*, and *current*.

NOTE This algorithm differs from the ordinary object OrdinaryDefineOwnProperty abstract operation algorithm only in invocation of [[GetOwnProperty]] in step 1.

8.4.3.4 [[Enumerate]] ()

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

8.4.3.5 [[OwnPropertyKeys]] ( )

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

8.4.3.6 StringCreate Abstract Operation

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

1. Let *A* be a newly created String exotic object.
2. Set *A*'s essential internal methods to the default ordinary object definitions specified in 8.3.
3. Set the [[HasOwnProperty]] internal method of *A* as specified in 8.4.3.1.

Deleted: .

Deleted: an

Deleted: *P* is a valid property key, either a String or a Symbol Object

Deleted: calling the default [[GetOwnProperty]] internal method (8.12.1) on

Deleted: with argument

Deleted: <#>ReturnIfAbrupt(*index*).¶

Deleted: <#>ReturnIfAbrupt(*absIntIndex*).¶

Deleted: ToString(abs(ToInteger(*P*))) is not the same val as

Deleted: ,

Deleted: <#>Let *index* be ToInteger(*P*).¶

Deleted: ¶
<#>8.4.3.3 [[GetP]] (P, Receiver)¶
When the [[GetP]] internal method of exotic String object *O* called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:¶
Assert: *P* is a valid property key, either a String or a Symbol Object.¶
Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.¶
ReturnIfAbrupt(*desc*).¶
If *desc* is **undefined**, then¶
Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.¶
ReturnIfAbrupt(*parent*).¶
If *parent* is **null**, then return **undefined**.¶
Return the result of calling the [[GetP]] internal methods of *parent* with arguments *P* and *Receiver*.¶
If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].¶
Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].¶
If *getter* is **undefined**, return **undefined**.¶
Return the result of calling the [[Call]] internal method of *getter* with *targetThis* as the *thisArgument* and an empty Lis *argumentsList*.¶
NOTE This algorithm differs from the ordinary object [[SetP]] algorithm only in invocation of [[GetOwnProperty in step 2.

Deleted: Let *extensible* be the result of calling the [[GetIsExtensible]] internal method of *O*.

Deleted: and [[IsExtensible]] in step 2

Comment [AWB1227]: TODO

Deleted: ¶
8.4.3.7 [[Keys]] ( )

Comment [AWB1228]: TODO

Deleted: When the [[Keys]] internal method of an exo String object *O* is called the following steps are taken:

Comment [AWB1329]: TODO

Deleted: (

4. Set the [[GetOwnProperty]] internal method of *A* as specified in 8.4.3.2.
5. Set the [[DefineOwnProperty]] internal method of *A* as specified in 8.4.3.3.
6. Set the [[Enumerate]] internal method of *A* as specified in 8.4.3.4.
7. Set the [[OwnPropertyKeys]] internal method of *A* as specified in 8.4.3.5.
8. Set the [[Prototype]] internal data property of *A* to *prototype*.
9. Set the [[Extensible]] internal data property of *A* to **true**.
10. Return *A*.

8.4.4 Symbol Exotic Objects

A *Symbol object* is an exotic object that may be used as a property key. Symbol exotic objects are always immutable and never observably reference any other object.

Exotic Symbol objects provide alternative definitions for all of the essential internal methods.

8.4.4.1 [[GetInheritance]] ( )

When the [[GetInheritance]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **null**.

8.4.4.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of an exotic Symbol object *O* is called with argument *V* the following steps are taken:

1. Assert: Either Type(*V*) is Object or Type(*V*) is Null.
2. Return **false**.

8.4.4.3 [[IsExtensible]] ( )

When the [[IsExtensible]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **false**.

8.4.4.4 [[PreventExtensions]] ( )

When the [[PreventExtensions]] internal method of an exotic Symbol object an exotic Symbol object *O* is called the following steps are taken:

1. Return **true**.

8.4.4.5 [[HasOwnProperty]] (P)

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

1. Return **false**.

8.4.4.6 [[GetOwnProperty]] (P)

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

1. Return **undefined**.

8.4.4.7 [[DefineOwnProperty]] (P, Desc)

When the [[DefineOwnProperty]] internal method of an exotic Symbol object *O* is called with property key *P* and property descriptor *Desc*, the following steps are taken:

1. Return **false**.

8.4.4.8 [[HasProperty]] (P)

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

1. Return **false**.

8.4.4.9 [[Get]] (P, Receiver)

When the [[Get]] internal method of an exotic Symbol 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. Return **undefined**.

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

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

1. Return **false**.

8.4.4.11 [[Invoke]] (P, ArgumentsList, Receiver)

When the [[Invoke]] internal method of an exotic Symbol object *O* is called with property key *P*, List *ArgumentsList*, and ECMAScipt language value *Receiver* the following steps are taken:

1. Throw a **TypeError** exception..

8.4.4.12 [[Delete]] (P)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Return **true**.

8.4.4.13 [[Enumerate]] ()

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

1. Return the result of CreateEmptyIterator().

8.4.4.14 [[OwnPropertyKeys]] ( )

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

1. Return the result of CreateEmptyIterator().

8.4.5 Exotic Arguments Objects

An *arguments object* is an exotic object whose array index properties map to the formal parameters bindings of an invocation of a non-strict function.

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

Exotic arguments objects provide alternative definitions for the following internal methods. All of the other exotic arguments object essential internal methods that are not defined below are as specified in 8.3.

8.4.6 Integer Indexed Exotic Objects

An *Integer Indexed object* is an exotic object that delegates [[Get]] and [[Set]] handling of integer property keys to methods of the object.

Integer Indexed exotic objects initially have the same internal data properties as ordinary objects.

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 8.3.

8.4.6.1 [[HasOwnProperty]] (P)

When the [[HasOwnProperty]] 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 data.
3. If Type(*P*) is String, then
   a. Let *intIndex* be ToInteger(*P*).
   b. Assert: *intIndex* is not an abrupt completion.
   c. If SameValue(ToString(*intIndex*), *P*) is **true**, then
      i. If *intIndex* < 0, then return **false**.
      ii. Let *length* be the value of *O*'s [[ArrayLength]] internal data property.
      iii. If *length* is **undefined**, then throw a **TypeError** exception.
      iv. If *intIndex* ≥ *length*, then return **false**.
4. Return the result of calling the ordinary object [[HasOwnProperty]] internal method (8.3.5) on *O* with argument *P*.

8.4.6.2 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of an Integer Indexed exotic object *S* 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 data.
3. If Type(*P*) is String, then
   a. Let *intIndex* be ToInteger(*P*).
   b. Assert: *intIndex* is not an abrupt completion.
   c. If SameValue(ToString(*intIndex*), *P*) is **true**, then
      i. Let *value* be the result of IntegerIndexedElementGet (*O*, *intIndex*).
      ii. ReturnIfAbrupt(*value*).
      iii. If *value* is **undefined**, then return **undefined**.
      iv. Let *writable* be **true** if the integer indexed properties of *O* are writable and **false** if they are not.
      v. Return a Property Descriptor { [[Value]]: *value*, [[Enumerable]]: **true**, [[Writable]]: *writable*, [[Configurable]]: **false** }.
4. Return the result of OrdinaryGetOwnProperty(*O*, *P*).

Comment [AWB630]: TODO

Deleted: <#>8.4.4 Exotic Symbol Obje
An *Symbol object* is an exotic object that may be used as a property key. Symbol exotic objects are unique in that they are always immutable and never observably reference a other object.¶
Exotic String objects have the a single interna data properties named [[Private]] that is set when the object is created and never modifie Exotic Symbol objects provide alternative definitions for all of the essential internal methods. ¶
<#>8.4.4.1 [[GetInheritance]] ( )¶
When the [[GetInheritance]] internal method o an exotic Symbol object *O* is called the following steps are taken:¶
<#>Return **null**.¶
<#>8.4.4.2 [[SetInheritance]] (V)¶
When the [[SetInheritance]] internal method o an exotic Symbol object *O* is called with argument *V* the following steps are taken:¶
<#>Assert: Either Type(*V*) is Object or Type(*V* Null.¶
<#>Return **false**.¶
<#>8.4.4.3 [[HasIntegrity]] (Level)[[IsExtensible]] ( )¶
When the [[HasIntegrityIsExtensible]] interna method of an exotic Symbol object *O* is calle the following steps are taken:¶
<#>Assert: *Level* is one of "**nonextensible** "**sealed**", or "**frozen**".¶
<#>Return **falsetrue**.¶

Deleted: <#>8.4.14 [[Freeze]] ( )¶

Deleted: invocation

Deleted: s

Deleted: property and a [[BuiltinBrand]] internal data property whose value is

Comment [AWB1231]: TODO move arguments internal methods here.

Deleted: Typed

Deleted:

Deleted: Delegation

Deleted: Array

Comment [AWB1332]: Issue: does the TypedArray spec./WEbIDL specs require tha

Deleted: *Delegation*

Deleted: that

Deleted: P

Deleted: P

Deleted: array index

Deleted: Delegation

Deleted: Exotic

Deleted: Delegation

Deleted: exotic

Deleted: Delegation

Comment [AWB1633]: TODO: need to formaize this.

8.4.6.3 [[DefineOwnProperty]] (P, Desc)

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

Comment [AWB1634]: TODO: need to formaize this

8.4.6.4 [[Get]] (P, Receiver)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. If Type(*P*) is String and if *SameValue*(*O*, *Receiver*) is **true**, then
   a. Let *intIndex* be ToInteger(*P*).
   b. Assert: *intIndex* is not an abrupt completion.
   c. If SameValue(ToString(*intIndex*), *P*) is **true**, then
      i. Return the result of IntegerIndexedElementGet (*O*, *intIndex*).
3. Return the result of calling the default ordinary object [[Get]] internal method (8.3.7) on *O* passing *P* and *Receiver* as arguments.

Comment [AWB1635]: TODO: need to formaize this

Deleted: P

Deleted: P

Deleted: exotic

Deleted: i

Deleted: delegation

Deleted:

Deleted: and

Deleted: an array index

Deleted: <#>Let *args* be a new List containing *int*
intIndexP.¶

Deleted: Invoke

Deleted: @@elementGet,

Deleted: *args*

Deleted:)

Deleted: P

Deleted: P

Deleted: P

Deleted: an

Deleted: exotic i

Deleted: delegation

Deleted:

Deleted: and

Deleted: an array index

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

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. If Type(*P*) is String and if SameValue(*O*, *Receiver*) is **true**, then
   a. Let *intIndex* be ToInteger(*P*).
   b. Assert: *intIndex* is not an abrupt completion.
   c. If SameValue(ToString(*intIndex*, *P*)) is **true**, then

i. Return the result of ToBoolean(IntegerIndexedElementSet (O, intIndex, V)).

4. Return the result of calling the default ordinary object [[Set]] internal method (8.3.7) on *O* passing *P*, *V*, and *Receiver* as arguments.

8.4.6.6 [[Enumerate]] ()

8.4.6.7 [[OwnPropertyKeys]] ()

8.4.6.8 IntegerIndexedObjectCreate Abstract Operation

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

11. Let *A* be a newly created ECMAScript object.
12. Set *A*'s essential internal methods to the default ordinary object definitions specified in 8.3.
13. Set the [[Get]] internal method of *A* as specified in 8.4.6.1.
14. Set the [[Set]] internal method of *A* as specified in 8.4.6.2.
15. Set the [[Prototype]] internal data property of *A* to *prototype*.
16. Set the [[Extensible]] internal data property of *A* to **true**.
17. Return *A*.

8.4.6.9 IntegerIndexedElementGet (O, index) Abstract Operation

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

8.4.6.10 IntegerIndexedElementSet (O, index, value) Abstract Operation

1. Assert: Type(*index*) is Number and *index* is an integer.
2. Let *O* be the result of calling ToObject with the **this** value as its argument.
3. ReturnIfAbrupt(*O*).
4. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
5. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
6. If *buffer* is **undefined**, then throw a **TypeError** exception.
7. Let *length* be the value of *O*'s [[ArrayLength]] internal data property.
8. Let *numValue* be ToNumber(*value*).
9. ReturnIfAbrupt(*numValue*).
10. If *index* < 0 or *index* ≥ *length*, then return *numValue*.
11. Let *offset* be the value of *O*'s [[ByteOffset]] internal data property.
12. Let *arrayTypeName* be the string value *O*'s [[TypedArrayName]] internal data property.
13. Let *elementSize* be the Number value of the Element Size value specified in Table 36 for *arrayTypeName*.
14. Let *indexedPosition* = (*index* × *elementSize*) + *offset*.
15. Let *elementType* be the string value of the Element Type value in Table 36 for *arrayTypeName*.
16. Let *status* be the result of SetValueInBuffer(*buffer*, *indexedPosition*, *elementType*, *numValue*).
17. ReturnIfAbrupt(*status*).
18. Return *numValue*.

Deleted: <#>Let *args* be a new List containing *index intIndexP* and *V*.¶
Deleted: Invoke
Deleted: @@elementSet,
Deleted: *args*)
Deleted: P
Comment [AWB1636]: TODO
Deleted: t
Deleted: Delegator
Deleted: Delegator
Deleted: (
Deleted: exotic
Deleted: Delegation
Deleted: P
Deleted: P

8.4.7 Built-in Function Objects

The function objects specified in Clause 15 may be implemented as either ordinary function objects whose behaviour is provided using ECMAScript code or as implementation provided exotic function objects whose behaviour is provided in some other manner. In either case, the effect of calling such functions must be that specified for each one in Clause 15.

If an implementation provided exotic object is used, the object must have the ordinary object behaviour specified in 8.3 except for [[Get]] and [[GetOwnProperty]] which must be as specified in 8.3.15. All such exotic function objects also have [[Prototype]] and [[Extensible]] internal data.

[[Call]] and [[Construct]]

8.4.7.1 CreateBuiltinFunction Abstract Operation

The abstract operation CreateBuiltinFunction takes a single argument, *steps*, that is a list of algorithm steps. It returns a bult-in function object created by following steps:

1. Let *func* be a new built-in function object, in the current Realm, that when called performs the action described by *steps*.
2. Return *func*.

8.5 Proxy Object Internal Methods and Internal Data Properties

A proxy object is an exotic object whose essential internal methods are partially implemented using ECMAScript code. Every proxy objects has an internal data property called [[ProxyHandler]]. The value of [[ProxyHandler]] is always an object, called the proxy's *handler object*. Methods of a handler object may be used to augment the implementation for one or more of the proxy object's internal methods. Every proxy object also has an internal data property called [[ProxyTarget]] whose value is either an object or the **null** value. This object is called the proxy's *target object*.

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

The [[ProxyHandler]] and [[ProxyTarget]] internal data properties of a proxy object are always initialised when the object is created and typically may not be modified. Some proxy objects are created in a manner that permits them to be subsequently *revoked*. When a proxy is revoked, its [[ProxyHander]] internal data property is set to a special revoked proxy handler object and its [[ProxyTarget]] internal data property is set to **null**.

Because proxy permit arbitrary ECMAScript code to be used to in the implementation of internal methods, it is possible to define a proxy object whose handler methods violates the invariants defined in 8.1.6.2. Some of the internal method invariants defined in 8.1.6.2 are essential integrity invariants. These invariants are explicitly enforced by the proxy internal methods specified in this section. An ECMAScript implementation must be robust in the presence of all possible invariant violations.

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

8.5.1 [[GetInheritance]] ( )

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

1. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "`getPrototypeOf`").
4. ReturnIfAbrupt(*trap*).

Deleted: 8.4.6

Deleted: s

Deleted: non-function

Deleted: P

Deleted: 9

Deleted: properties and a [[BuiltinBrand]] internal dat property whose value is **BuiltinFunction**

Comment [AWB1237]: TODO: need to talk about [[Call]] and [[Construct]] behaviour of chapter 15 native functions.

Deleted: I

Comment [AWB1638]: Perhaps this needs to be elaborated.

Deleted: usually

Deleted: s

Deleted: z

Deleted: that

Deleted: An ECMAScript implementation must be rob in the presence of such violations.

Deleted: proxy

5. If *trap* is **undefined**, then
   a. Return the result of calling the [[GetInheritance]] internal method of *target*.
6. Let *handlerProto* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*handlerProto*).
8. Let *targetProto* be the result of calling the [[GetInheritance]] internal method of *target*.
9. ReturnIfAbrupt(*targetProto*).
10. If SameValue(*handlerProto*, *targetProto*) is **false**, then throw a **TypeError** exception.
11. Return *handlerProto*.

NOTE [[GetInheritance]] for proxy objects enforces the following invariant:

- [[GetInheritance]] applied to the proxy object must return the same value as [[GetInheritance]] applied to the proxy object's target object.

8.5.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of an exotic Proxy object *O* is called with argument *V* the following steps are taken:

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

NOTE [[SetInheritance]] for proxy objects enforces the following invariant:

- If the target object is not extensible, the argument value must be the same as the result of [[GetInheritance]] applied to target object.

8.5.2 [[IsExtensible]] ( )

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

1. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "`isExtensible`").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
   a. Return the result of calling the [[IsExtensible]] internal method of *target*.
6. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. Let *booleanTrapResult* be ToBoolean(*trapResult*).
9. Let *targetResult* be the result of calling the [[IsExtensible]] internal method of *target*.
10. ReturnIfAbrupt(*targetResult*).
11. If SameValue(*booleanTrapResult*, *targetResult*) is **false**, then throw a **TypeError** exception.
12. Return *booleanTrapResult*.

Deleted: *trapResult*

Deleted: handler

Deleted: proxy

Deleted: <#>Set *extensibleTarget* to ToBoolean(*extensibleTarget*).¶

Deleted: <#>Let *getProtoTrap* be the result o GetMethod(*handler*, "`getPrototypeOf`").¶
<#>ReturnIfAbrupt(*getProtoTrap*).¶
<#>If *getProtoTrap* is **undefined**, then¶
<#>Return *trapResult*.¶
<#>Let *getProtoResult* be the result of calling *getProtoTrap* with *handler* as the **this** value and new List containing *target*.¶
<#>ReturnIfAbrupt(*getProtoResult*).¶

Deleted: *getProtoResult*

Deleted: After a

Deleted: [[SetInheritance]] call, [[GetInheritance]] applied to the proxy obje mustmay only return true if

Deleted: is now the valued return the sam value asfor

Deleted:

Deleted: the proxy object's

Deleted: handler

Deleted: HasIntegrity

Deleted:

Deleted: HasIntegrity

Deleted: proxy

Deleted: <#>If *Level* is "`nonextensible`" then let *trapName* be "`isExtensible`".¶
<#>Else if *Level* is "`sealed`", then let *trapNa* be "`isSealed`".¶
<#>Else *Level* is "`frozen`", so let *trapName* b "`isFrozen`".¶

Deleted: *trapName*

Deleted: HasIntegrity

Deleted:

Deleted: *proxyIsExtensible*

Deleted: *targetIsExtensible*

Deleted: HasIntegrity

Deleted: *targetIsExtensible*

Deleted: *proxyIsExtensible*

Deleted: *targetIsExtensible*

Deleted: *proxyIsExtensible*

NOTE [[IsExtensible]] for proxy objects enforces the following invariant:

- [[IsExtensible]] applied to the proxy object must return the same value as [[IsExtensible]] applied to the proxy object's target object with the same argument.

8.5.3 [[PreventExtensions]] ( )

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

1. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`preventExtensions`**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[PreventExtensions]] internal method of *target*.
6. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*.
7.
8. Let *booleanTrapResult* be ToBoolean(*trapResult*).
9. ReturnIfAbrupt(*booleanTrapResult*).
10. Let *targetIsExtensible* be the result of calling the [[IsExtensible]] internal method of *target*.
11. ReturnIfAbrupt(*targetIsExtensible*).
12. If *booleanTrapResult* is **true** and *targetIsExtensible* is **true**, then throw a **TypeError** exception.
13. Return *proxyIsExtensible*.

NOTE [[PreventExtensions]] for proxy objects enforces the following invariant:

- [[PreventExtensions]] applied to the proxy object only returns **true** if [[IsExtensible]] applied to the proxy object's target object is **false**.

8.5.4 [[HasOwnProperty]] (P)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`hasOwn`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[HasOwnProperty]] internal method of *target* with argument *P*.
7. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResult*).
9. Let *success* be ToBoolean(*trapResult*).
10. If *success* is **false**, then
    a. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
    b. ReturnIfAbrupt(*targetDesc*).
    c. If *targetDesc* is not **undefined**, then
        i. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
        ii. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
        iii. ReturnIfAbrupt(*extensibleTarget*).
        iv. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
11. Else *success* is **true**,
    a. Let *extensibleTarget* be the result of IsExtensible(*target*).
    b. ReturnIfAbrupt(*extensibleTarget*).
    c. If ToBoolean(*extensibleTarget*) is **true**, then return *success*.
    d. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
    e. ReturnIfAbrupt(*targetDesc*).

Deleted: handler
Deleted: Setntegrity
Deleted: proxy
Moved down [3]: ReturnIfAbrupt(*trapResult*).¶
Deleted: <#>Let *booleanTrapResult* be ToBoolean(*trapResult*).¶
Deleted: <#>Let *isTrap* be the result of GetMethod(*handler*, "**`isExtensible`**").¶
<#>ReturnIfAbrupt(*isTrap*).¶
<#>If *isTrap* is **undefined**, then¶
<#>Return NormalCompletion(empty).¶
<#>Let *isTrapResult* be the result of calling *isTrap* with *handler* as the **this** value and a new List containing *target*.¶
<#>ReturnIfAbrupt(*isTrapResult*).¶
Deleted: *proxyIsExtensible*
Deleted: *isT*
Deleted: .
Deleted: ¶
Moved (insertion) [3]
Deleted: *trapResult*
Deleted: SameValue(*proxyIsExtensible*
Deleted: ,
Deleted:)
Deleted: **false**
Deleted: *booleanTrapResult*
Deleted: NormalCompletion(empty)
Deleted: SetIntegrity]
Deleted: After a
Deleted: SetIntegrity
Deleted: must return the same
Formatted: Font: (Default) +Headings CS (Times New Roman), Bold, Complex Script Font: +Headings CS (Times New Roman), Bold
Deleted: value as
Deleted: HasIntegrity
Formatted: Font: (Default) +Headings CS (Times New Roman), Bold, Complex Script Font: +Headings CS (Times New Roman), Bold
Deleted:
Deleted: with the same argument
Deleted: [[PreventExtensions]] call, [[IsExtensible]] applied to the proxy object must return the same val
Deleted: proxy
Deleted: *P* is a valid property key, either a String or a
Deleted: <#>If *targetDesc* is **undefined**, then¶
Deleted: calling the [[
Deleted:]] internal method of
Deleted: **false**
Deleted: throw a **TypeError** exception

f. If *targetDesc* is **undefined**, then throw a **TypeError** exception.

12. Return *success*.

NOTE [[HasOwnProperty]] for proxy objects enforces the following invariants:

- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as a own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exists as a own property of the target object and the target object is not extensible.

8.5.5 [[GetOwnProperty]] (P)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "`getOwnPropertyDescriptor`").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
   a. Return the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
7. Let *trapResultObj* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResultObj*).
9. If Type(*trapResultObj*) is neither Object or Undefined, then throw a **TypeError** exception.
10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *trapResultObj* is **undefined**, then
    a. If *targetDesc* is **undefined**, then return **undefined**.
    b. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
    c. Let *extensibleTarget* be the result of IsExtensible(*target*).
    d. ReturnIfAbrupt(*extensibleTarget*).
    e. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
    f. Return **undefined**.
13. Let *extensibleTarget* be the result of IsExtensible(*target*).
14. ReturnIfAbrupt(*extensibleTarget*).
15. Set *extensibleTarget* to ToBoolean(*extensibleTarget*).
16. Let *resultDesc* be ToPropertyDescriptor(*trapResultObj*).
17. ReturnIfAbrupt(*resultDesc*).
18. Call CompletePropertyDescriptor(*resultDesc*, *targetDesc*).
19. Let *valid* be the result of IsCompatiblePropertyDescriptor (*extensibleTarget*, *resultDesc*, *targetDesc*).
20. If *valid* is **false**, then throw a **TypeError** exception.
21. If *resultDesc*.[[Configurable]] is **false**, then
    a. If *targetDesc* is **undefined** or *targetDesc*.[[Configurable]] is **true**, then
       i. Throw a **TypeError** exception.
22. 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 a own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exists as a own property of the target object and the target object is not extensible.
- A property cannot be reported as non-configurable, if it does not exists as a own property of the target object or if it exists as a configurable own property of the target object.
- The result of [[GetOwnProperty]] can be applied to the target object using [[DefineOwnPropery]] and will not throw an exception.

Deleted: proxy

Deleted: Assert: *P* is a valid property key, eith a String or a Symbol Object.

Deleted: u

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Comment [AWB1239]: Note the result descriptor defaults are set to the values in the targetDesc (if there is one) rather than the normal defaults. This is a change from the w spec.

Deleted: a

Deleted: not

Deleted: and

Comment [AWB1240]: The resultDesc carries a reference to the original descriptor returned by the trap. A copy is not made and missing attribute properties are not added to

This is a change from the wiki spec.

8.5.6 [[DefineOwnProperty]] (P, Desc)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "`defineProperty`").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
   a. Return the result of calling the [[DefineOwnProperty]] internal method of *target* with arguments *P* and *Desc*.
7. Let *descObj* be FromPropertyDescriptor(*Desc*).
8. NOTE If *Desc* was originally generated from an object using ToPropertyDescriptor, then *descObj* will be that original object.
9. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*, *P*, and *descObj*.
10. ReturnIfAbrupt(*trapResult*).
11. If ToBoolean(*trapResult*) is **false**, then return **false**.
12. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
13. ReturnIfAbrupt(*targetDesc*).
14. Let *extensibleTarget* be the result of IsExtensible(*target*).
15. ReturnIfAbrupt(*extensibleTarget*).
16. Set *extensibleTarget* to ToBoolean(*extensibleTarget*).
17. If *Desc* has a [[Configurable]] field and if *Desc*.[[Configurable]] is **false**, then
    a. Let *settingConfigFalse* be **true**.
18. Else let *settingConfigFalse* be **false**.
19. If *targetDesc* is **undefined**, then
    a. If *extensibleTarget* is **false**, then throw a **TypeError** exception.
    b. If *settingConfigFalse* is **true**, then throw a **TypeError** exception.
20. Else *targetDesc* is not **undefined**,
    a. If IsCompatiblePropertyDescriptor(*extensibleTarget*, *Desc* , *targetDesc*) is **false**, then throw a **TypeError** exception.
    b. If *settingConfigFalse* is **true** and *targetDesc*.[[Configurable]] is **true**, then throw a **TypeError** exception.
21. Return **true**.

NOTE [[DefineOwnProperty]] for proxy objects enforces the following invariants:

- A property cannot be added, if the target object is not extensible.
- A property cannot be added as or modified to be non-configurable, if it does not exists as a non-configurable own property of the target object.
- A property may not be non-configurable, if is corresponding configurable property of the target object exists.
- If a property has a corresponding target object property then apply the property descriptor of the property to the target object using [[DefineOwnPropery]] will not throw an exception.

Deleted: calling the [[

Deleted:]] internal method of

Deleted: *Desc*.[[Configurable]]

Deleted: **false**

Deleted: *Desc*.[[Configurable]] is **false**

Deleted: Get

8.5.7 [[HasProperty]] (P)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "`has`").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
   a. Return the result of calling the [[HasProperty]] internal method of *target* with argument *P*.
7. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResult*).

9. Let *success* be ToBoolean(*trapResult*).
10. If *success* is **false**, then
    a. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
    b. ReturnIfAbrupt(*targetDesc*).
    c. If *targetDesc* is not **undefined**, then
        i. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
        ii. Let *extensibleTarget* be the result of IsExtensible(*target*).
        iii. ReturnIfAbrupt(*extensibleTarget*).
        iv. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
11. Return *success*.

NOTE [[HasProperty]] for proxy objects enforces the following invariants:

- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as a own property of the target object and the target object is not extensible.

8.5.8 [[Get]] (P, Receiver)

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "`get`").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[Get]] internal method of *target* with arguments *P* and *Receiver*.
7. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*, *P*, and *Receiver*.
8. ReturnIfAbrupt(*trapResult*).
9. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
10. ReturnIfAbrupt(*targetDesc*).
11. If *targetDesc* is not **undefined**, then
    a. If IsDataDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Writable]] is **false**, then
        i. If SameValue(*trapResult*, *targetDesc*.[[Value]]) is **false**, then throw a **TypeError** exception.
    b. If IsAccessorDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Get]] is **undefined**, then
        i. If *trapResult* is not **undefined**, then throw a **TypeError** exception.
12. Return *trapResult*.

NOTE [[Get]] for proxy objects enforces the following invariants:

- The value reported for a property must be the same as the value of the corresponding target object property if the target object property is a non-writable, non-configurable data property.
- The value reported for a property must be **undefined** if the corresponding target object property is non-configurable accessor property that has **undefined** as its [[Get]] attribute.

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

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

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "`set`").
5. ReturnIfAbrupt(*trap*).

Deleted: calling the [[
Deleted:]] internal method of
Deleted: P
Deleted: P
Deleted: proxy
Deleted: Assert: *P* is a valid property key, eith a String or a Symbol Object
Deleted: .
Deleted: P
Deleted: P
Deleted: .
Deleted: P
Deleted: P
Deleted: proxy
Deleted: Assert: *P* is a valid property key, eith a String or a Symbol Object.

6. If *trap* is **undefined**, then
    a. Return the result of calling the [[Set]] internal method of *target* with arguments *P*, *V*, and *Receiver*.
7. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*, *P*, *V*, and *Receiver*.
8. ReturnIfAbrupt(*trapResult*).
9. If ToBoolean(*trapResult*) is **false**, then return **false**.
10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *targetDesc* is not **undefined**, then
    a. If IsDataDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Writable]] is **false**, then
        i. If SameValue(*V*, *targetDesc*.[[Value]]) is **false**, then throw a **TypeError** exception.
    b. If IsAccessorDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false**, then
        i. If *targetDesc*.[[Set]] is **undefined**, then throw a **TypeError** exception.
13. Return **true**.

NOTE [[Set]] for proxy objects enforces the following invariants:

- Cannot change the value of a property to be different from the value of the corresponding target object property if the corresponding target object property is a non-writable, non-configurable data property.
- Cannot set the value of a property if the corresponding corresponding target object property is a non-configurable accessor property that has **undefined** as its [[Set]] attribute.

Deleted: P

Deleted: P

Deleted: o

8.5.10 [[Invoke]] (P, ArgumentsList, Receiver)

When the [[Invoke]] internal method of an exotic Proxy object *O* is called with property key *P*, List *ArgumentsList*, and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: IsPropertyKey(*P*) is **true**..
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**invoke**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[Invoke]] internal method of *target* with arguments *P*, *ArgumentsList*, and *Receiver*.
7. Let *argArray* be the result of CreateArrayFromList(*ArgumentsList*).
8. Return the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*, *P*, *argArray*, and *Receiver*.

NOTE There are no invariants enforced for [[Invoke]].

8.5.11 [[Delete]] (P)

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

Deleted: proxy

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**deleteProperty**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[Delete]] internal method of *target* with argument *P*.
7. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResult*).
9. If ToBoolean(*trapResult*) is **false**, then return **false**.
10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *targetDesc* is **undefined**, then return **true**.

Deleted: Assert: *P* is a valid property key, either a String a Symbol Object.

13. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
14. Return **true**.

NOTE [[Delete]] for proxy objects enforces the following invariant:

- A property cannot be deleted, if it exists as a non-configurable own property of the target object.

8.5.12 [[Enumerate]] ()

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

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

TODO: we may need to add a lot of additional invariant checking here according to the wiki spec. But maybe it really isn't necessary. Tomvc response: I think it may be possible to waive the extra invariant checks for [[Enumerate]]. It's not a crucial primitive. My reasoning is that [[Enumerate]] deals with both own and inherited properties, and we don't really enforce any invariants on inherited properties. So I guess it's ok if the invariants for [[Enumerate]] are weakened.
Do note that this is a bit inconsistent with the way we treat internal methods like [[HasProperty]], [[GetP]] and [[SetP]]: these also deal with own and inherited properties, but still enforce invariants on own properties.

9. Return *trapResult*.

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

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

8.5.13 [[OwnPropertyKeys]] ( )

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

1. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**ownKeys**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
   a. Return the result of calling the [[OwnPropertyKeys]] internal method of *target*.
6. Let *trapResult* be the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. If Type(*trapResult*) is not Object, then throw a **TypeError** exception.
9. TODO: we may need to add a lot of additional invariant checking here according to the wiki spec. But maybe it really isn't necessary
10. Return *trapResult*.

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

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

8.5.14 [[Call]] (thisArgument, argumentsList)

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

1. Let *handler* be the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* be the value of the [[ProxyTarget]] internal data property of *O*.

Deleted: *d*

Deleted: <#>8.5.10 [[DefineOwnPrope y]] (P, Desc)¶
When the [[DefineOwnProperty]] internal method of proxy object *O* is called with prope key *P* and property descriptor *Desc*, the following steps are taken:¶
<#>Assert: *P* is a valid property key, either a String or a Symbol Object.¶
<#>Let *handler* the value of the [[ProxyHandle internal data property of *O*. ¶
<#>Let *target* the value of the [[ProxyTarget]] internal data property of *O*. ¶
<#>Let *trap* be the result of GetMethod(*handle* "**defineProperty**").¶
<#>ReturnIfAbrupt(*trap*).¶
<#>If *trap* is **undefined**, then¶
<#>Return the result of calling the [[DefineOwnProperty]] internal method of *target* with arguments *P* and *Desc*.¶
<#>Let *descObj* be FromPropertyDescriptor(*Desc*).¶
<#>NOTE If *Desc* was originally generated from an object using ToPropertyDescriptor, th *descObj* will be that original object.¶
<#>Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*, *P*, and *descObj*.¶
<#>ReturnIfAbrupt(*trapResult*).¶
<#>If ToBoolean(*trapResult*) is **false**, then retu **false**.¶
<#>Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* w argument *P*.¶
<#>ReturnIfAbrupt(*targetDesc*).¶
<#>Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.¶
<#>ReturnIfAbrupt(*extensibleTarget*).¶
<#>Set *extensibleTarget* to ToBoolean(*extensibleTarget*),¶
<#>If *targetDesc* is **undefined**, then¶
<#>If *extensibleTarget* is **false**, then throw a **TypeError** exception.¶
<#>If *Desc*.[[Configurable]] is **false**, then throw a **TypeError** exception.¶
<#>Else *targetDesc* is not **undefined**,¶

Deleted: of

Deleted: proxy

Comment [AWB1241]: TODO.

Comment [AWB1242]: TODO

Deleted: <#>8.5.12 [[Keys]] ( )¶
When the [[Keys]] internal method of proxy object *O* is called the following steps are take

Deleted: proxy

Deleted: getO

Deleted: Property

Deleted: Names

Comment [AWB1243]: TODO

Deleted: <#>8.5.14 [[Freeze]] ( )¶
When the [[Freeze]] internal method of an exotic Proxy proxy object *O* is called the following steps are taken:¶

3. Let *trap* be the result of GetMethod(*handler*, "**apply**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
   a. Return the result of calling the [[Call]] internal method of *target* with arguments *thisArgument* and *argumentsList*.
6. Let *argArray* be the result of CreateArrayFromList(*argumentsList*).
7. Return the result of calling the [[Call]] internal method of *trap* with *handler* as the **this** value and a new List containing *target*, *thisArgument*, and *argArray*.

NOTE An Proxy exotic object only has a [[Call]] internal method if the initial value of its [[ProxyTarget]] internal data property is an object that has a [[Call]] internal method.

8.5.15 [[Construct]] Internal Method

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

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

NOTE 1 An Proxy exotic object only has a [[Construct]] internal method if the initial value of its [[ProxyTarget]] internal data property 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 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.

9.1 Type Conversion and Testing

The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; they can accept a value of any ECMAScript language type or of a Completion Record value. But no other specification types are used with these operations.

9.1.1 ToPrimitive

The abstract operation ToPrimitive takes an input *argument* and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to Table 15:

Deleted: Return

Deleted: ¶

Deleted: L

Deleted: runtime system

Deleted: These abstract operations are not a part of t language; they are defined here to aid the specification the semantics of the language

Deleted: .

Deleted: that is,

Deleted: ,

Deleted: but

Deleted: not of

Table 15 — ToPrimitive Conversions

Input Type	*Result*
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToPrimitive(*argument*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *argument* (no conversion).
Null	Return *argument* (no conversion).
Boolean	Return *argument* (no conversion).
Number	Return *argument* (no conversion).
String	Return *argument* (no conversion).
Object	Perform the steps following this table.

When the *InputType* is Object, the following steps are taken:

1. If *PreferredType* was not passed, let *hint* be "**`default`**".
2. Else if *PreferredType* is hint String, let *hint* be "**`string`**".
3. Else *PreferredType* is hint Number, let *hint* be "**`number`**".
4. Let *exoticToPrim* be the result of Get(*argument*, @@ToPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. If IsCallable(*exoticToPrim*) is **false**, then throw a **TypeError** exception.
    b. Let *result* be the result of calling the [[Call]] internal method of *exoticToPrim*, with *argument* as *thisArgument* and a List containing *hint* as *argumentsList*.
    c. ReturnIfAbrupt(*result*).
    d. If *result* is an ECMAScript language value and Type(*result*) is not Object, then return *result*.
    e. Else, throw a **TypeError** exception.
7. If *hint* is "**`default`**" then, let *hint* be "**`number`**".
8. Return the result of OrdinaryToPrimitive(*argument*,*hint*).

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

1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either "**`string`**" or "**`number`**".
3. If *hint* is "**`string`**", then
    a. Let *tryFirst* be "**`toString`**".
    b. Let *trySecond* be "**`valueOf`**".
4. Else,
    a. Let *tryFirst* be "**`valueOf`**".
    b. Let *trySecond* be "**`toString`**".
5. Let *first* be the result of Get(*O*, *tryFirst*).
6. ReturnIfAbrupt(*first*).
7. If IsCallable(*first*) is **true** then,
    a. Let *result* be the result of calling the [[Call]] internal method of *first*, with *O* as *thisArgument* and an empty List as *argumentsList*.
    b. ReturnIfAbrupt(*result*).
    c. If *result* is an ECMAScript language value and Type(*result*) is not Object, then return *result*.
8. Let *second* be the result of Get(*O*, *trySecond*).
9. ReturnIfAbrupt(*second*).
10. If IsCallable(*second*) is **true** then,
    a. Let *result* be the result of calling the [[Call]] internal method of *second*, with *O* as *thisArgument* and an empty argument list.
    b. ReturnIfAbrupt(*result*).
    c. If *result* is an ECMAScript language value and Type(*result*) is not Object, then return *result*.
11. Throw a **TypeError** exception.

Deleted: The result equals
Deleted:
Deleted: the *input* argumen
Deleted: t
Deleted: The result equals
Deleted: the *input* argument
Deleted: The result equals
Deleted: the *input* argument
Deleted: The result equals
Deleted: the *input* argument
Deleted: The result equals
Deleted: the *input* argument
Deleted: following
Deleted: given
Deleted: :¶
<#>Let *default* be the result of calling the [[DefaultValue]] internal method of *argument*, passing the optional hint *PreferredType*.¶
<#>Return ToPrimitive(*default*).¶
Return a default value for the Object. The default value of an object is retrieved by callin the [[DefaultValue]] internal method of the object, passing the optional hint *PreferredTyp* The behaviour of the [[DefaultValue]] internal method is defined by this specification for all native ECMAScript objects in 8.12.8.
Deleted: *O*
Deleted: *toString*
Deleted: *O*
Deleted: be
Deleted: *O*
Deleted: s
Deleted: *First*
Deleted: <#>Else, throw a **TypeError** exception.¶

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@ToPrimitive method. Of the objects defined in this specification only Date objects (see 15.9.6) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

9.1.2 ToBoolean

The abstract operation ToBoolean converts its *argument* to a value of type Boolean according to Table 16:

Table 16 — ToBoolean Conversions

Argument Type	***Result***
Completion Record	If *argument* is an abrupt completion, return the argument. Otherwise return ToBoolean(*argument*.[[value]])
Undefined	Return **false**
Null	Return **false**
Boolean	Return the input argument (no conversion).
Number	Return **false** if the argument is **+0**, **–0**, or **NaN**; otherwise return **true**.
String	Return **false** if the argument is the empty String (its length is zero); otherwise return **true**.
Object	Return **true**

Deleted: The result equals

Deleted: The result is

Deleted: the result is

Deleted: The result is

Deleted: the result is

9.1.3 ToNumber

The abstract operation ToNumber converts its *argument* to a value of type Number according to Table 17:

Table 17 — ToNumber Conversions

Deleted:

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

Deleted: The result is

Deleted: the

Deleted: The result is

Deleted: the

Deleted: The result equals

Deleted: the *input*

Deleted: *input*

9.1.3.1 ToNumber Applied to the String Type

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

Syntax

StringNumericLiteral **:::**
    StrWhiteSpace$_{opt}$
    StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::**
    StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **:::**
    WhiteSpace
    LineTerminator

StrNumericLiteral **:::**
    StrDecimalLiteral
    HexIntegerLiteral

StrDecimalLiteral **:::**
    StrUnsignedDecimalLiteral
    + *StrUnsignedDecimalLiteral*
    - *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral **:::**
    `Infinity`
    DecimalDigits **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
    . *DecimalDigits ExponentPart*$_{opt}$
    DecimalDigits ExponentPart$_{opt}$

DecimalDigits **:::**
    DecimalDigit
    DecimalDigits DecimalDigit

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

ExponentPart **:::**
    ExponentIndicator SignedInteger

ExponentIndicator **::: one of**
    `e E`

SignedInteger **:::**
    DecimalDigits
    + *DecimalDigits*
    - *DecimalDigits*

HexIntegerLiteral **:::**
    `0x` *HexDigit*
    `0X` *HexDigit*
    HexIntegerLiteral HexDigit

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

NOTE Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (see 7.8.3):

- A *StringNumericLiteral* may be preceded and/or followed by white space and/or line terminators.
- A *StringNumericLiteral* that is decimal may have any number of leading **`0`** digits.
- A *StringNumericLiteral* that is decimal may be preceded by **+** or **-** to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only white space is converted to **+0**.
- **`Infinity`** and **`-Infinity`** are recognised as a *StringNumericLiteral* but not as a *NumericLiteral*.

Deleted: z

Runtime Semantics

The conversion of a String to a Number value is similar overall to the determination of the Number value for a numeric literal (see 7.8.3), but some of the details are different, so the process for converting a String numeric literal to a value of Number type is given here in full. This value is determined in two steps: first, a mathematical value (MV) is derived from the String numeric literal; second, this mathematical value is rounded as described below.

- The MV of *StringNumericLiteral* **:::** [empty] is 0.
- The MV of *StringNumericLiteral* **:::** *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* **:::** $StrWhiteSpace_{opt}$ *StrNumericLiteral* $StrWhiteSpace_{opt}$ is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *StrNumericLiteral* **:::** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* **:::** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* **:::** **+** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* **:::** **-** *StrUnsignedDecimalLiteral* is the negative of the MV of *StrUnsignedDecimalLiteral*. (Note that if the MV of *StrUnsignedDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this signless mathematical zero to a floating-point **+0** or **−0** as appropriate.)
- The MV of *StrUnsignedDecimalLiteral* **:::** `Infinity` is 10^{10000} (a value so large that it will round to **+∞**).
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits* **.** is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits* **.** *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n}), where n is the number of characters in the second *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits* **.** *ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits* **.** *DecimalDigits ExponentPart* is (the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n})) times 10^{e}, where n is the number of characters in the second *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* **:::** **.** *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where n is the number of characters in *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* **:::** **.** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* **:::** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *DecimalDigits* **:::** *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* **:::** *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* **:::** *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* **:::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:::** **+** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:::** **-** *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* **:::** `0` or of *HexDigit* **:::** `0` is 0.
- The MV of *DecimalDigit* **:::** `1` or of *HexDigit* **:::** `1` is 1.
- The MV of *DecimalDigit* **:::** `2` or of *HexDigit* **:::** `2` is 2.
- The MV of *DecimalDigit* **:::** `3` or of *HexDigit* **:::** `3` is 3.
- The MV of *DecimalDigit* **:::** `4` or of *HexDigit* **:::** `4` is 4.
- The MV of *DecimalDigit* **:::** `5` or of *HexDigit* **:::** `5` is 5.
- The MV of *DecimalDigit* **:::** `6` or of *HexDigit* **:::** `6` is 6.
- The MV of *DecimalDigit* **:::** `7` or of *HexDigit* **:::** `7` is 7.
- The MV of *DecimalDigit* **:::** `8` or of *HexDigit* **:::** `8` is 8.
- The MV of *DecimalDigit* **:::** `9` or of *HexDigit* **:::** `9` is 9.
- The MV of *HexDigit* **:::** `a` or of *HexDigit* **:::** `A` is 10.
- The MV of *HexDigit* **:::** `b` or of *HexDigit* **:::** `B` is 11.
- The MV of *HexDigit* **:::** `c` or of *HexDigit* **:::** `C` is 12.
- The MV of *HexDigit* **:::** `d` or of *HexDigit* **:::** `D` is 13.
- The MV of *HexDigit* **:::** `e` or of *HexDigit* **:::** `E` is 14.
- The MV of *HexDigit* **:::** `f` or of *HexDigit* **:::** `F` is 15.
- The MV of *HexIntegerLiteral* **:::** `0x` *HexDigit* is the MV of *HexDigit*.

- The MV of *HexIntegerLiteral* ::: `0x` *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* ::: *HexIntegerLiteral HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a String numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non white space character in the String numeric literal is '–', in which case the rounded value is –0. Otherwise, the rounded value must be the Number value for the MV (in the sense defined in 8.5), unless the literal includes a *StrUnsignedDecimalLiteral* and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is *significant* if it is not part of an *ExponentPart* and

- it is not **0**; or
- there is a nonzero digit to its left and there is a nonzero digit, not in the *ExponentPart*, to its right.

9.1.4 ToInteger

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

1. Let *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, return **+0**.
4. If *number* is **+0**, **–0**, +∞, or –∞, return *number*.
5. Return the result of computing sign(*number*) × floor(abs(*number*)).

9.1.5 ToInt32: (Signed 32 Bit Integer)

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

1. Let *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, **+0**, **–0**, +∞, or –∞, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int32bit* be *int* modulo 2^{32}.
6. If *int32bit* ≥ 2^{31}, return *int32bit* – 2^{32}, otherwise return *int32bit*.

Deleted: *posInt*

Deleted: *

Deleted: *posInt*

Deleted: ; that is, a finite integer value *k* of Number type with positive sign and less than 2^{32} in magnitude such that the mathematical difference of *posInt int* and *k* is mathematically an integer multiple of 2^{32}

Deleted: is greater than or equal to

NOTE Given the above definition of ToInt32:

- The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToInt32(ToUint32(*x*)) is equal to ToInt32(*x*) for all values of *x*. (It is to preserve this latter property that +∞ and –∞ are mapped to **+0**.)
- ToInt32 maps **–0** to **+0**.

9.1.6 ToUint32: (Unsigned 32 Bit Integer)

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

Deleted: ion

1. Let *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, +0, –0, +∞, or –∞, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int32bit* be *int* modulo 2^{32}.
6. Return *int32bit*.

Deleted: *posInt*

Deleted: *posInt*

Deleted: ; that is, a finite integer value *k* of Number type with positive sign and less than 2^{32} in magnitude such that the mathematical difference of *posInt int* and *k* is mathematically an integer multiple of 2^{32}

NOTE Given the above definition of ToUint32:

Deleted: ToUInt32

- Step 6 is the only difference between ToUint32 and ToInt32.
- The ToUint32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToUint32(ToInt32(x)) is equal to ToUint32(x) for all values of x. (It is to preserve this latter property that +∞ and –∞ are mapped to **+0**.)
- ToUint32 maps –**0** to **+0**.

Deleted: 5

9.1.7 ToUint16: (Unsigned 16 Bit Integer)

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

1. Let *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, +0, –0, +∞, or –∞, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int16bit* be *int* modulo 2^{16}.
6. Return *int16bit*.

Deleted: *posInt*

Deleted: *posInt*

Deleted: ; that is, a finite integer value *k* of Number type with positive sign and less than 2^{16} in magnitude such that the mathematical difference of *posInt int* and *k* is mathematically an integer multiple of 2^{16}

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**.

9.1.8 ToString

The abstract operation ToString converts its *argument* to a value of type String according to Table 18:

Table 18 — ToString Conversions

Argument Type	*Result*
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]])
Undefined	`"undefined"`
Null	`"null"`
Boolean	If *argument* is **true**, then return `"true"`. If *argument* is **false**, then return `"false"`.
Number	See 9.8.1.
String	Return *argument* (no conversion)
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

Deleted: *the*

Deleted: the result is

Deleted: the

Deleted: the result is

Deleted: the input

Deleted: *input*

9.1.8.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number m to String format as follows:

1. If m is **NaN**, return the String `"NaN"`.
2. If m is **+0** or **–0**, return the String `"0"`.
3. If m is less than zero, return the String concatenation of the String `"-"` and ToString($–m$).
4. If m is +∞, return the String `"Infinity"`.
5. Otherwise, let n, k, and s be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, 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 k digits of the decimal representation of s (in order, with no leading zeroes), followed by $n–k$ occurrences of the character '`0`'.

Deleted: infinity

7. If $0 < n \le 21$, return the String consisting of the most significant n digits of the decimal representation of s, followed by a decimal point '`.`', followed by the remaining $k-n$ digits of the decimal representation of s.
8. If $-6 < n \le 0$, return the String consisting of the character '`0`', followed by a decimal point '`.`', followed by $-n$ occurrences of the character '`0`', followed by the k digits of the decimal representation of s.
9. Otherwise, if $k = 1$, return the String consisting of the single digit of s, followed by lowercase character '`e`', followed by a plus sign '`+`' or minus sign '`-`' according to whether $n-1$ is positive or negative, followed by the decimal representation of the integer abs($n-1$) (with no leading zeroes).
10. Return the String consisting of the most significant digit of the decimal representation of s, followed by a decimal point '.', followed by the remaining $k-1$ digits of the decimal representation of s, followed by the lowercase character '`e`', followed by a plus sign '`+`' or minus sign '`-`' according to whether $n-1$ is positive or negative, followed by the decimal representation of the integer abs($n-1$) (with no leading zeroes).

NOTE 1 The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard:

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

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

> Otherwise, let n, k, and s be integers such that $k \ge 1$, $10^{k-1} \le s < 10^k$, 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:

> Gay, David M. Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis, Manuscript 90-10. AT&T Bell Laboratories (Murray Hill, New Jersey). November 30, 1990. Available as http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz. Associated code available as http://netlib.sandia.gov/fp/dtoa.c and as http://netlib.sandia.gov/fp/g_fmt.c and may also be found at the various **`netlib`** mirror sites.

9.1.9 ToObject

The abstract operation ToObject converts its *argument* to a value of type Object according to Table 19:

Table 19 — ToObject Conversions

Argument Type	*Result*
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToObject(*argument*.[[value]])
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal data property is set to the value of *argument*. See 15.6 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal data property is set to the value of *argument*. See 15.7 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal data property is set to the value of *argument*. See 15.5 for a description of String objects.
Object	Return *argument* (no conversion).

9.1.10 ToPropertyKey

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

Deleted: http://cm.bell-labs.com/netlib/fp/dtoa.c.gz

Deleted: http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz

Deleted: Create

Deleted: PrimitiveValue

Deleted: the

Deleted: Create

Deleted: PrimitiveValue

Deleted: *the*

Deleted: Create

Deleted: PrimitiveValue

Deleted: the

Deleted: The result is

Deleted: the input

1. ReturnIfAbrupt(*argument*).
2. If *argument* is an exotic Symbol Object, then
   a. Return *argument*.
3. Return ToString(*argument*).

9.2 Testing and Comparison Operations

9.2.1 CheckObjectCoercible

The abstract operation CheckObjectCoercible throws an error if its argument is a value that cannot be converted to an Object using ToObject. It is defined by Table 20:

Table 20 — CheckObjectCoercible Results

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return CheckObjectCoercible(*argument*.[[value]])
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return *argument*
Number	Return *argument*
String	Return *argument*
Object	Return *argument*

9.2.2 IsCallable

The abstract operation IsCallable determines if its *argument*, which must be an ECMAScript language value or a Completion Record, is a callable function Object according to Table 21:

Table 21 — IsCallable Results

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return IsCallable(*argument*.[[value]])
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return **false**.
Number	Return **false**.
String	Return **false**.
Object	If *argument* has a [[Call]] internal method, then return **true**, otherwise return **false**.

9.2.3 SameValue(x, y)

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

1. ReturnIfAbrupt(*x*).
2. ReturnIfAbrupt(*y*).
3. If Type(*x*) is different from Type(*y*), return **false**.
4. If Type(*x*) is Undefined, return **true**.
5. If Type(*x*) is Null, return **true**.
6. If Type(*x*) is Number, then
   a. If *x* is NaN and *y* is NaN, return **true**.
   b. If *x* is +0 and *y* is -0, return **false**.
   c. If *x* is -0 and *y* is +0, return **false**.
   d. If *x* is the same Number value as *y*, return **true**.

Deleted: Type(

Deleted:)

Deleted: Object

Deleted: Symbol

Deleted: <#>If *argument* has a [[NativeBrand]] inter property whose value is NativePrivateNameis an exotic StringSymbol object, then¶

Deleted: <#>9.1.11 ToPositiveInteger¶
The abstract operation ToInteger converts its argumen an integral numeric value. This abstract operation functions as follows:¶
<#>Let *number* be the result of calling ToNumber on the input argument.¶
<#>ReturnIfAbrupt(*number*).¶
<#>If *number* is **NaN**, return **+0**.¶
<#>If *number* is +∞, or –∞, return *number*.¶
<#>If *number* ≤ **0**, return **+0**.¶
<#>Return the result of computing floor(*number*).¶

Deleted: i

Deleted: *argument*

Deleted: *argument*

Deleted: *the*

Deleted: object

Deleted: The

Deleted: Algorithm

Deleted: <#>If *x* is an Completion Record, let *x* be *x*.[[value]]. ¶
<#>If *y* is an Completion Record, let *y* be *y*.[[value]]. ¶

Deleted: .

e. Return **false**.

7. If Type(*x*) is String, then
   a. If *x* and *y* are exactly the same sequence of code units (same length and same code units in corresponding positions) return **true**; otherwise, return **false**.
8. If Type(*x*) is Boolean, then
   a. If *x* and *y* are both **true** or both **false**, then return **true**; otherwise, return **false**.
9. Return **true** if *x* and *y* are the same Object value. Otherwise, return **false**.

9.2.4 SameValueZero(x, y)

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

1. ReturnIfAbrupt(*x*).
2. ReturnIfAbrupt(*y*).
3. If Type(*x*) is different from Type(*y*), return **false**.
4. If Type(*x*) is Undefined, return **true**.
5. If Type(*x*) is Null, return **true**.
6. If Type(*x*) is Number, then
   a. If *x* is NaN and *y* is NaN, return **true**.
   b. If *x* is +0 and *y* is -0, return **true**.
   c. If *x* is -0 and *y* is +0, return **true**.
   d. If *x* is the same Number value as *y*, return **true**.
   e. Return **false**.
7. If Type(*x*) is String, then
   a. If *x* and *y* are exactly the same sequence of code units (same length and same code units in corresponding positions) return **true**; otherwise, return **false**.
8. If Type(*x*) is Boolean, then
   a. If *x* and *y* are both **true** or both **false**, then return **true**; otherwise, return **false**.
9. Return **true** if *x* and *y* are the same Object value. Otherwise, return **false**.

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

9.2.5 IsConstructor

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

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

9.2.6 IsPropertyKey

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

1. ReturnIfAbrupt(*argument*).
2. If Type(*argument*) is String, return **true**.
3. If Type(*argument*) is Object and *argument* is an exotic Symbol object, return **true**.
4. Return **false**.

9.2.7 IsExtensible (O)

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

1. Assert: Type(*O*) is Object.

Deleted: return **true** i
Deleted: characters
Deleted: characters
Deleted: return **true** i
Deleted: refer to
Deleted: object
Formatted: Note
Deleted: Callable
Deleted: value
Deleted: operation is called with
Deleted:
Deleted: *O*

2. Return the result of calling the [[IsExtensible]] internal method of *O*.

9.3 Operations on Objects

9.3.1 Get (O, P)

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

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Return the result of calling the [[Get]] internal method of *O* passing *P* and *O* as the arguments.

9.3.2 Put (O, P, V, Throw)

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

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Assert: Type(*Throw*) is Boolean.
4. Let *success* be the result of calling the [[Set]] internal method of *O* passing *P*, *V*, and *O* as the arguments.
5. ReturnIfAbrupt(*success*).
6. If *success* is **false** and *Throw* is **true**, then throw a **TypeError** exception.
7. Return *success*.

9.3.3 CreateOwnDataProperty (O, P, V)

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

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Assert: *O* does not have an own property whose key is *P*.
4. Let *newDesc* be the Property Descriptor {[[Value]]: *V*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
5. Return the result of calling the [[DefineOwnProperty]] internal method of *O* passing *P* and *newDesc* as arguments.

NOTE This abstract operation creates a property whose attributes as set to the same defaults used for properties created by the ECMAScript language assignment operator.

9.3.4 DefinePropertyOrThrow (O, P, desc)

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

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Let *success* be the result of calling the [[DefineOwnProperty]] internal method of *O* passing *P* and *desc* as arguments.
4. ReturnIfAbrupt(*success*).
5. If *success* is **false**, then throw a **TypeError** exception.
6. Return *success*.

Deleted: Let *notExtensible* be
Deleted: *globalObject*
Formatted: Don't add space between paragraphs of t same style
Deleted: HasIntegrity
Deleted: with argument "**nonextensible**"
Deleted: ¶
<#>ReturnIfAbrupt(*notExtensible*).¶
If *notExtensib le* is **true**, then return false**false**; else return **true**.
Deleted: Property Access
Deleted: ,
Deleted: Either
Deleted: Type
Deleted: **String**
Deleted: or Type(*O*) is Object and *O* is a Symbol object.
Deleted: P
Deleted: n
Deleted: ,
Deleted: Either Type(*P*) is String or Type(*O*) is Object a *O* is a Symbol object
Deleted: P
Deleted: ,
Deleted: Either Type(*P*) is String or Type(*O*) is Object a *O* is a Symbol object
Deleted: .
Deleted: <#>Let *noteExtensible* be the result of calling [[GetIsHasExtensibleIntegrity]] internal method of *O* with argument "**nonextensible**".¶
<#>ReturnIfAbrupt(*enotExtensible*).¶
<#>If *notEextensible* is **falsetrue**, then return **false**.¶
Deleted: .
Deleted:
Formatted: Note
Deleted:
Deleted: record
Deleted: s
Deleted: Either Type(*P*) is String or Type(*O*) is Object a *O* is a Symbol object

9.3.5 DeletePropertyOrThrow (O, P)

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

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Let *success* be the result of calling the [[Delete]] internal method of *O* passing *P* as the argument.
4. ReturnIfAbrupt(*success*).
5. If *success* is **false**, then throw a **TypeError** exception.
6. Return *success*.

9.3.6 HasProperty (O, P)

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

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

9.3.7 GetMethod (O, P)

The abstract operation GetMethod is used to get the value of an specific property of an object when the value of the property is expected to be a function. The operation is called with arguments *O* and *P* where *O* is the object, *P* is the property key. This abstract operation performs the following steps:

1. Assert: Type(*O*) is Object.
2. Assert: IsPropertyKey(*P*) is **true**.
3. Let *func* be the result of calling the [[Get]] internal method of *O* passing *P* and *O* as the arguments.
4. ReturnIfAbrupt(*func*).
5. If *func* is **undefined**, then return **undefined**.
6. If IsCallable(*func*) is **false**, then throw a **TypeError** exception.
7. Return *func*.

9.3.8 Invoke(O,P, [args])

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

1. Assert: *P* is a valid property key.
2. If *args* was not passed, then let *args* be a new empty List.
3. If Type(*O*) is Object then,
   a. Let *base* be *O*.
4. Else,
   a. Let *base* be ToObject(*O*).
5. ReturnIfAbrupt(*base*).
6. Return the result of calling the [[Invoke]] internal method of *base* passing arguments *P*, *args*, and *O*.

9.3.9 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"`.

Deleted: ,
Deleted: Either Type(*P*) is String or Type(*O*) Object and *O* is a Symbol object
Deleted: ,
Deleted: Either Type(*P*) is String or Type(*O*) Object and *O* is a Symbol object
Deleted: <#>Let *obj* be *O*.¶
<#>Repeat,¶
<#>If *obj* is **null**, then return **false**.¶
<#>If Type(*obj*) is not Object, then throw a **TypeError** exception.¶
<#>Let *has* be the result of calling the [[HasOwnProperty]] internal method of *obj* argument *P*.¶
<#>ReturnIfAbrupt(*has*).¶
<#>If *has* is **true**, return *has*.¶
<#>Set *obj* to the result of calling the [[GetInheritance]] internal method of *obj*.¶
Deleted: IfAbrupt(*obj*).
Deleted: ,
Deleted: Either Type(*P*) is String or Type(*O*) Object and *O* is a Symbol object
Deleted: P
Deleted:
Deleted: *V*
Deleted: ,
Deleted: Invoke
Deleted: ion
Deleted: abstract
Deleted: ,
Deleted: *obj*
Deleted: *obj*
Deleted: <#>Let *func* be the result of calling [[GetMethod]] internal method of(*obj*, passing as) the argument.¶
<#>ReturnIfAbrupt(*func*).¶
<#>If IsCallable(*func*) is **falseundefined**, throw **TypeError** exception.¶
Deleted: Call
Deleted: *func*
Deleted: *O*
Deleted: as *thisArgument* the **this** value and
Deleted: argument list
Deleted: as
Deleted: *argumentsList*
Deleted: MakeObjectSecure
Deleted: immutable
Deleted: MakeObjectSecure
Deleted: If the Boolean argument *immutabl* **true** all own data properties are also made n
Deleted: ,
Deleted: <#>Assert: Type(*immutable*) is

3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *O*.
4. ReturnIfAbrupt(*keys*).
5. Let *pendingException* be **undefined**.
6. If *level* is "`sealed`", then
    a. Repeat for each element *k* of *keys*,
        i. Let *status* be the result of DefinePropertyOrThrow(*O*, *k*, PropertyDescriptor{ [[Configurable]]: **false**}).
        ii. If *status* is an Abrupt Completion, then
            1. If *pendingException* is **undefined**, then set *pendingException* to *status*.
7. Else *level* is "`frozen`",
    a. Repeat for each element *k* of *keys*,
        i. Let *status* be the result of calling the [[GetOwnProperty]] internal method of *O* with *k*.
        ii. If *status* is an Abrupt Completion, then
            1. If *pendingException* is **undefined**, then set *pendingException* to *status*.
        iii. Else,
            1. Let *currentDesc* be *status*.[[value]].
            2. If *currentDesc* is not **undefined**, then
                a. If IsAccessorDescriptor(*currentDesc*) is **true**, then
                    i. Let *desc* be PropertyDescriptor{[[Configurable]]: **false**}.
                b. Else,
                    i. Let *desc* be PropertyDescriptor { [[Configurable]]: **false**, [[Writable]]: **false** }.
                c. Let *status* be the result of DefinePropertyOrThrow(*O*, *k*, *desc*).
                d. If *status* is an Abrupt Completion, then
                    i. If *pendingException* is **undefined**, then set *pendingException* to *status*.
8. If *pendingException* is not **undefined**, then return *pendingException*.
9. Return the result of calling the [[PreventExtensions]] internal method of *O*.

9.3.10 TestIntegrityLevel (O, level)

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

1. Assert: Type(*O*) is Object.
2. Assert: *level* is either "`sealed`" or "`frozen`".
3. Let *status* be the result of IsExtensible(*O*).
4. ReturnIfAbrupt(*status*).
5. If *status* is **true**, then return **false**
6. NOTE If the object is extensible, none of its properties are examined.
7. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *O*.
8. ReturnIfAbrupt(*keys*).
9. Let *pendingException* be **undefined**.
10. Let *configurable* be **false**.
11. Let *writable* be **false**.
12. Repeat for each element *k* of *keys*,
    a. Let *status* be the result of calling the [[GetOwnProperty]] internal method of *O* with *k*.
    b. If *status* is an Abrupt Completion, then
        i. If *pendingException* is **undefined**, then set *pendingException* to *status*.
        ii. Let *configurable* be **true**.
    c. Else,
        i. Let *currentDesc* be *status*.[[value]].
        ii. If *currentDesc* is not **undefined**, then
            1. Set *configurable* to *configurable* logically ored with *currentDesc*.[[Configurable]].
            2. If IsDataDescriptor(*currentDesc*) is **true**, then
                a. Set *writable* to *writable* logically ored with *currentDesc*.[[Writable]].
13. If *pendingException* is not **undefined**, then return *pendingException*.
14. If *level* is "`frozen`" and *writable* is **true**, then return **false**.
15. If *configurable* is **true**, then return **false**.

Deleted: *immutable*
Deleted: **false**
Deleted: TestIfSecureObject
Deleted: immutable
Deleted: TestIfSecureObject
Deleted: If the Boolean argument *immutable* is **true** a check is also made to determine whether all own data properties are non-writable.
Deleted: ,
Deleted: <#>Assert: Type(*immutable*) is Boolean.¶
Deleted: calling the [[
Deleted: Get
Deleted:]] internal method of
Deleted:
Deleted: <#>If *immutable* is **false**, then¶
Deleted: *immutable*
Deleted: **true**

16. Return **true**.

9.3.11 CreateArrayFromList (elements)

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

1. Assert: *elements* is a List whose elements are all ECMAScript language values.
2. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
3. Let *n* be 0.
4. For each element *e* of *elements*
   a. Let *status* be the result of CreateOwnDataProperty(*array*, ToString(*n*), *e*).
   b. Assert: *status* is **true**.
   c. Increment *n* by 1.
5. Return *array*.

9.3.12 CreateListFromArrayLike (obj)

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

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

9.3.13 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 data property, then
   a. Let *BC* be the value of *C's* [[BoundTargetFunction]] internal data property.
   b. Return the result of instanceofOperator(*O*,*BC*) (see 11.8).
3. If Type(*O*) is not Object, return **false**.
4. Let *P* be the result of Get(*C*, `"prototype"`).
5. ReturnIfAbrupt(*P*).
6. If Type(*P*) is not Object, throw a **TypeError** exception.
7. Repeat
   a. Set *O* to the result of calling the [[GetInheritance]] internal method of *O* with no arguments.
   b. ReturnIfAbrupt(*O*).
   c. If *O* is `null`, return **false**.
   d. If SameValue(*P*, *O*) is **true**, return **true**.

9.3.14 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

Deleted: ,
Deleted: Call
Deleted: *the call in step 4.a will never resu* *an abrupt completion*
Formatted: Font: Italic, Complex Script Fon Italic
Deleted: used
Deleted: ,
Deleted: r
Deleted: a
Deleted: the
Deleted: O
Deleted: abstract operation
Deleted:) with *C* and *O* as arguments
Deleted: an

prototype property, if it exists. Otherwise the supplied default is used for [[Prototype]]. This abstract operation performs the following steps:

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

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

9.3.15 OrdinaryCreateFromConstructor (constructor, intrinsicDefaultProto, internalDataList)

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

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

10 Executable Code and Execution Contexts

10.1 Types of Executable Code

There are four types of ECMAScript executable code:

- *Global code* is source text that is treated as an ECMAScript *Script*. The global code of a particular *Script* does not include any source text that is parsed as part of a *FunctionBody*, *GeneratorBody*, *ConciseBody*, *ClassBody*, or *ModuleBody*.
- *Eval code* is the source text supplied to the built-in **eval** function. More precisely, if the parameter to the built-in **eval** function is a String, it is treated as an ECMAScript *Script*. The eval code for a particular invocation of **eval** is the global code portion of that *Script*.
- *Function code* is source text that is parsed to supply the value of the [[Code]] internal data property (see 8.3.15) of function and generator objects. The *function code* of a particular function or generator does not include any source text that is parsed as the function code of a nested *FunctionBody*, *GeneratorBody*, *ConciseBody*, or *ClassBody*.
- *Module code* is source text that is parse code that is provided as a *ModuleBody*. It is the code that is directly evaluated when a module is initialised. The module code of a particular module does not include any source text that is parsed as part of a nested *FunctionBody*, *GeneratorBody*, *ConciseBody*, *ClassBody*, or *ModuleBody*.

Deleted: *F*

Deleted: be

Deleted: <#>If Type(*constructor*) is not Object, then thr a **TypeError** exception.¶ <#>Let *proto* be the result of Get(*construtor*, **"prototype"**).¶ <#>ReturnIfAbrupt(*proto*).¶ <#>If Type(*proto*) is not Object, then¶ <#>If *constructor* has a [[Realm]] internal data propert let *realm* be *F's* [[Realm]].¶ <#>Else,¶ <#>Let *ctx* be the running execution context.¶ <#>Let *realm* be be *ctx's* Realm.¶ <#>Let *proto* be *realm's* intrinsic object named *intrinsicDefaultProto*.¶

Deleted: with argument

Deleted:

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

Deleted: three

Deleted: five

Deleted: *Program*

Deleted: *Program*

Deleted: *Program*

Deleted: *Program*

Comment [AWB744]: TODO: may need an addition reference to the [[Code]] of a generator.

Deleted: 13.56

Deleted: as part of a *FunctionBody*.

Deleted: *FunctionBody*

Deleted: part

Deleted: function or generator

Deleted: *FunctionBody*

Deleted: .

Deleted: <#>*Generator code* is source text that is parsed to supply the value of the [[Code]] internal dat property (see 13.5) of generator objects. The *genera code* of a particular generator does not include any source text that is parsed as the function code of a nested *FunctionBody*, *ConciseBody*, or *ClassBody*. All generator code is also considered to be function code but only function code that is defined within a generat is generator code. ¶

Deleted: z

Deleted: .

Deleted: *Function code* also denotes the source text supplied when using the built-in **Function** object as constructor. More precisely, the last parameter provid to the **Function** constructor is converted to a String and treated as the *FunctionBody*. If more than one parameter is provided to the **Function** constructor, a parameters except the last one are converted to Strin

NOTE Function code is generally provided as the bodies of Function Definitions (13.1), Arrow Function Definitions (13.2), Method Definitions (13.3) and Generator Definitions (13.4). Function code is also derived from the last argument to the Function constructor (15.3).

10.1.1 Strict Mode Code

An ECMAScript *Script* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the four types of ECMAScript code are referred to as module code, strict global code, strict eval code, and strict function code. Code is interpreted as strict mode code in the following situations:

- Global code is strict global code if it begins with a Directive Prologue that contains a Use Strict Directive (see 14.1).
- Module code is always strict code.
- All code contained in a *ClassBody* is strict code.
- Eval code is strict eval code if it begins with a Directive Prologue that contains a Use Strict Directive or if the call to eval is a direct call (see 15.1.2.1.1) to the eval function that is contained in strict mode code.
- Function code that is part of a *FunctionDeclaration*, *FunctionExpression*, or accessor *PropertyDefinition* is strict function code if its *FunctionDeclaration*, *FunctionExpression*, or *PropertyDefinition* is contained in strict mode code or if the function code begins with a Directive Prologue that contains a Use Strict Directive.
- Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a String that when processed as a *FunctionBody* begins with a Directive Prologue that contains a Use Strict Directive.

10.1.2 Non-ECMAScript Functions

An ECMAScript implementation may support the evaluation of function objects whose evaluative behaviour is expressed in some implementation defined form of executable code other than via ECMAScript code. Whether a function object is an ECMAScript code function or a non-ECMAScript function is not semantically observable from the perspective of an ECMAScript code function that calls or is called by such a non-ECMAScript function.

10.2 Lexical Environments

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

An *Environment Record* records the identifier bindings that are created within the scope of its associated Lexical Environment.

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

A *global environment* is a Lexical Environment which does not have an outer environment. The global environment's outer environment reference is **null**. A global environment's environment record may be

Deleted: d

Deleted: Generator code is provided as the bodies of Generator Definitions 13.4 and Generator Expressions (11.????).

Deleted: *Program*

Deleted: three

Deleted: *PropertyAssignment*

Deleted: *PropertyAssignment*

Deleted: ¶

Deleted: <#>Unless specified otherwise, extended code (10.21.12) is also considere strict mode code.¶

Deleted: <#>10.1.2 Extended Code¶ *Extended code* is any any code contained in ECMAScript *Program* syntactic unit that conta occurraences of lexical or syntactic productio defined subsequent to the Fifth Edition of the ECMAScript specification. Code is interprete as extended code in the following situations:¶ <#>Global code is extended global code if is contained in an ECMAScript *Program* syntactic unit that has been designated as a extended *Program* unit in an implementation defined manner or if ???.¶ <#>Eval code is extended eval code if the to eval is a direct call (see 15.1.2.1.1) to the eval function that is contained in extended mode code or if it begins with ???. ¶ <#>Function code that is part of a *FunctionDeclaration*, *FunctionExpression*, or accessor *PropertyAssignment* is extended function code if its *FunctionDeclaration*, *FunctionExpression*, or *PropertyAssignment* is contained in extended mode code or if the function code begins with ???.¶ <#>Function code that is supplied as the la argument to the built-in Function constructo strict function code if the last argument is a String that when processed as a *FunctionBo* begins with ???.¶ The term "strict code" includesis used to designate both actual strict mode code and extended code while the term "extended co isluccoonly designates only used to to mear actual extended code.

Deleted: *WithStatement*

Deleted: execution

prepopulated with identifier bindings and includes an associated *global object* whose properties provide some of the global environment's identifier bindings. This global object is the value of a global environment's `this` binding. As ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modified.

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

Deleted: the

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.

10.2.1 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.

Deleted: *Program* and

For specification purposes Environment Record values can be thought of as existing in a simple object-oriented hierarchy where Environment Record is an abstract class with three concrete subclasses, declarative environment record, object environment record, and global environment record. Function environment records are a subclass of declarative environment record. The abstract class includes the abstract specification methods defined in Table 22. These abstract methods have distinct concrete algorithms for each of the concrete subclasses.

Deleted: two

Deleted: and

Table 22 — Abstract Methods of Environment Records

Method	Purpose
HasBinding(N)	Determine if an environment record has a binding for an identifier. Return **true** if it does and **false** if it does not. The String value *N* is the text of the identifier.
CreateMutableBinding(N, D)	Create a new but uninitialised mutable binding in an environment record. The String value *N* is the text of the bound name. If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted.
CreateImmutableBinding(N)	Create a new but uninitialised immutable binding in an environment record. The String value N is the text of the bound name.
InitialiseBinding(N,V)	Set the value of an already existing but uninitialised binding in an environment record. The String value N is the text of the bound name. V is the value for the binding and is a value of any ECMAScript language type.
SetMutableBinding(N,V, S)	Set the value of an already existing mutable binding in an environment record. The String value *N* is the text of the bound name. *V* is the value for the binding and may be a value of any ECMAScript language type. *S* is a Boolean flag. If *S* is **true** and the binding cannot be set throw a **TypeError** exception. *S* is used to identify strict mode references.
GetBindingValue(N,S)	Returns the value of an already existing binding from an environment record. The String value *N* is the text of the bound name. *S* is used to identify strict mode references. If *S* is **true** and the binding does not exist or is uninitialised throw a **ReferenceError** exception.
DeleteBinding(N)	Delete a binding from an environment record. The String value *N* is the text of the bound name If a binding for *N* exists, remove the binding and return **true**. If the binding exists but cannot be removed return **false**. If the binding does not exist return **true**.
HasThisBinding()	Determine if an environment record establishes a **`this`** binding. Return **true** if it does and **false** if it does not.
HasSuperBinding()	Determine if an environment record establishes a **`super`** method binding. Return **true** if it does and **false** if it does not.
WithBaseObject ()	If this environment record is associated with a **`with`** statement, return the with object. Otherwise, return **undefined**.

Deleted: InitializeBinding

Deleted: CreateVarBinding(N, D)

Deleted: ImplicitThisValue

Deleted: Returns the value to use as the thi[s] value on calls to function objects that are obtained as binding values from this environment record.

10.2.1.1 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.

10.2.1.1.1 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**.

10.2.1.1.2 CreateMutableBinding (N, D)

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

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

10.2.1.1.3 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name *N* that is uninitialised. A binding must not already exist in this environment record for *N*.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create an immutable binding in *envRec* for *N* and record that it is uninitialised.

10.2.1.1.4 InitialiseBinding (N,V)

The concrete Environment Record method InitialiseBinding for declarative environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. An uninitialised binding for *N* must already exist.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* must have an uninitialised binding for *N*.
3. Set the bound value for *N* in *envRec* to *V*.
4. Record that the binding for *N* in *envRec* has been initialised.

10.2.1.1.5 SetMutableBinding (N,V,S)

The concrete Environment Record method SetMutableBinding for declarative environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. A binding for *N* must already exist. If the binding is an immutable binding, a **TypeError** is thrown if S is **true**.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* must have a binding for *N*.
3. If the binding for *N* in *envRec* has not yet been initialised throw a **ReferenceError** exception.
4. Else if the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
5. Else this must be an attempt to change the value of an immutable binding so if *S* is **true** throw a **TypeError** exception.
6. Return NormalCompletion(empty).

10.2.1.1.6 GetBindingValue(N,S)

The concrete Environment Record method GetBindingValue for declarative environment records simply returns the value of its bound identifier whose name is the value of the argument *N*. The binding must already exist. If *S* is **true** and the binding is an uninitialised immutable binding throw a **ReferenceError** exception.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* has a binding for *N*.
3. If the binding for *N* in *envRec* is an uninitialised binding, then
   a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.

Deleted: If it does not have such a binding, r

Deleted: to the value **undefined**

Deleted:

Deleted: and

Deleted: set its bound value to **undefined**

Comment [AWB45]: This probably needs a D option argument, just like createMutableEnvironment

Deleted: InitializeBinding

Deleted: InitializeBinding

Deleted: <#>If the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.¶
Else i

Deleted: f

Deleted: z

Deleted: if

Deleted: **undefined**

Deleted: immutable

4. Else
   a. Return the value currently bound to *N* in *envRec*.

10.2.1.1.7 DeleteBinding (N)

The concrete Environment Record method DeleteBinding for declarative environment records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. If *envRec* does not have a binding for the name that is the value of *N*, return **true**.
3. If the binding for *N* in *envRec* cannot be deleted, return **false**.
4. Remove the binding for *N* from *envRec*.
5. Return **true**.

10.2.1.1.8 HasThisBinding ()

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

1. Return **false**.

10.2.1.1.9 HasSuperBinding ()

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

1. Return **false**.

10.2.1.1.10 WithBaseObject()

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

1. Return **undefined**.

10.2.1.2 Object Environment Records

Each object environment record is associated with an object called its *binding object*. An object environment record binds the set of identifier names that directly correspond to the property names of its binding object. Property names that are not an *IdentifierName* are not included in the set of bound identifiers. Both own and inherited properties are included in the set regardless of the setting of their [[Enumerable]] attribute. Because properties can be dynamically added and deleted from objects, the set of identifiers bound by an object environment record may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a result of such a side-effect are considered to be a mutable binding even if the Writable attribute of the corresponding property has the value **false**. Immutable bindings do not exist for object environment records.

Object environment records created for `with` statements (12.10) can provide their binding object as an implicit this value for use in function calls. The capability is controlled by a *withEnvironment* Boolean value that is associated with each object environment record. By default, the value of *withEnvironment* is **false** for any object environment record.

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

10.2.1.2.1 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. Return the result of HasProperty(*bindings*, *N*).

Deleted: ,

Deleted: return

Deleted: is

Deleted: <#>10.2.1.1.6 CreateVarBin g (N, D)¶
<#>The concrete Environment Record method CreateVarBinding for declarati environment records performs the sam action as the CreateMutableBinding concrete method of the same environment Record.¶
<#>Let *envRec* be the declarative environment record for which the meth was invoked.¶
<#>Return the result of calling *envRe* CreateMutableBinding concrete metho with arguments *N* and *D*. ¶

Deleted: ImplicitThisValue

Deleted: ImplicitThisValue

Comment [AWB46]: This probably needs a option argument, just like createMutableEnvironment

Deleted: <#>10.2.1.1.710 CreateImmu bleBinding (N)¶
<#>The concrete Environment Record method CreateImmutableBinding for declarative environment records create a new immutable binding for the name that is initialised to the value undefined A binding must not already exist in this environment record for *N*.¶
<#>Let *envRec* be the declarative environment record for which the meth was invoked.¶
<#>Assert: *envRec* does not already have a binding for *N*.¶
<#>Create an immutable binding in *envRec* for *N* and record that it is uninitialised.¶
<#>10.2.1.1.811 InitializeImmutableBi ng (N,V)¶
<#>The concrete Environment Record method InitializeImmutableBinding for declarative environment records is use to set the bound value of the current binding of the identifier whose name is the value of the argument *N* to the valu of argument *V*. An uninitialised immuta binding for *N* must already exist. ¶
<#>Let *envRec* be the declarative environment record for which the meth was invoked.¶

Deleted: be configured to

Deleted: This capability is used to specify th behaviour of With Statement (12.10) induced bindings.

Deleted: *provideThis*

Deleted: *provideThis*

Deleted: calling

Deleted: the [[HasProperty]] internal method c

Deleted: , passing

Deleted: as the property name

10.2.1.2.2 **CreateMutableBinding (N, D)**

The concrete Environment Record method CreateMutableBinding for object environment records creates in an environment record's associated binding object a property whose name is the String value and initialises it to the value **undefined**. A property named *N* must not already exist in the binding object. If Boolean argument *D* is provided and has the value **true** the new property's [[Configurable]] attribute is set to **true**, otherwise it is set to **false**.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Assert: The result of HasProperty(*bindings*, *N*) is **false**.
4. If *D* is **true** then let *configValue* be **true** otherwise let *configValue* be **false**.
5. Return the result of DefinePropertyOrThrow(*bindings*, *N*, Property Descriptor {[[Value]]:**undefined**, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *configValue*}).

10.2.1.2.3 **CreateImmutableBinding (N)**

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

10.2.1.2.4 **InitialiseBinding (N,V)**

The concrete Environment Record method InitialiseBinding for object environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. An uninitialised binding for *N* must already exist.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Assert: *envRec* must have an uninitialised binding for *N*.
3. Record that the binding for *N* in *envRec* has been initialised.
4. Call the SetMutableBinding concrete method of *envRec* with *N*, *V*, and **false** as arguments.

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

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

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Return the result of Put(*bindings*, *N*, *V*, and *S*).

10.2.1.2.6 **GetBindingValue(N,S)**

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

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Let *value* be the result of HasProperty(*bindings*, *N*).
4. ReturnIfAbrupt(*value*).
5. If *value* is **false**, then
   a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
6. Return the result of Get(*bindings*, *N*).

Deleted: calling the [[HasProperty]] internal method of
Deleted: passing
Deleted: as the property name
Deleted: ,
Deleted:
Deleted: Call
Deleted: the
Deleted: [[DefineOwnProperty]] internal method of
Deleted: passing
Deleted: ,
Deleted: , and **true** as arguments
Comment [AWB47]: This probably needs a D option argument, just like createMutableEnvironment
Deleted: InitializeBinding
Deleted: InitializeBinding
Deleted: should
Deleted: Call
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: calling the [[HasProperty]] internal method of
Deleted: passing
Deleted: as the property name
Deleted: calling the [[
Deleted:]] internal method of
Deleted: passing
Deleted: for the argument

10.2.1.2.7 DeleteBinding (N)

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

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Return the result of calling the [[Delete]] internal method of *bindings* passing *N* as the argument.

10.2.1.2.8 HasThisBinding ()

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

1. Return **false**.

10.2.1.2.9 HasSuperBinding ()

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

1. Return **false**.

10.2.1.2.10 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**.

10.2.1.3 Function Environment Records

A function environment record is a declarative environment record that is used to represent the outer most scope of a function that provides a **`this`** binding. In addition to its identifier bindings, a function environment record contains the **`this`** value used within its scope. If such a function references **`super`**, its function environment record also contains the state that is used to perform **`super`** method invocations from within the function.

Function environment records store their **`this`** binding as the value of their *thisValue*. If the associated function references **`super`**, the environment record stores in *HomeObject* the object that the function is bound to as a method and in *MethodName* the property key used for unqualified super invocations from within the function. The default value for *HomeObject* and *MethodName* is **undefined**.

Methods environment records support all of Declarative Environment Record methods listed in Table 22 and share the same specifications for all of those methods except for HasThisBinding and HasSuperBinding. In addition, declarative environment records support the methods listed in Table 23:

Table 23 — Additional Methods of Function Environment Records

Method	*Purpose*
GetThisBinding()	Return the value of this environment record's **`this`** binding.
GetSuperBase()	Return the object that is the base for **`super`** property accesses bound in this environment record. The object is derived from this environment record's HomeObject binding. If the value is Empty, return **undefined**.
GetMethodName()	Return the value of this environment record's MethodName binding.

Deleted: ,
Deleted: and false
Deleted: s
Deleted: <#>10.2.1.12.6 CreateVarBi ng (N, D)¶ <#>The concrete Environment Record method CreateVarBinding for object environment records performs the sam action as the CreateMutableBinding concrete method of the same environment Record.¶ <#>Let *envRec* be the declarative environment record for which the meth was invoked.¶ <#>Return the result of calling *envRe* CreateMutableBinding concrete metho with arguments *N* and *D*. ¶
Deleted: ImplicitThisValue
Deleted: ImplicitThisValue
Deleted: *provideThis*
Deleted: *provideThis*
Deleted: Method
Deleted: method
Deleted: method
Deleted: method
Deleted: Method
Deleted: and Table 20
Deleted: those
Deleted: j
Deleted:
Deleted: Table 21
Deleted: Method
Deleted: Home
Deleted: Object
Deleted: value
Deleted: of

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

10.2.1.3.1 HasThisBinding ()

Function Environment Records always provide a **`this`** binding.

1. Return **true**.

10.2.1.3.2 HasSuperBinding ()

1. If this environment record's *HomeObject* has the value Empty, then return **false**. Otherwise, return **true**.

10.2.1.3.3 GetThisBinding ()

1. Return the value of this environment record's *thisValue*.

10.2.1.3.4 GetSuperBase ()

1. Let *home* be the value of this environment record's *HomeObject*.
2. If *home* has the value Empty, then return **undefined**.
3. Assert Type(*home*) is Object.
4. Return the result of calling *home's* [[GetInheritance]] internal method.

10.2.1.3.5 GetMethodName ()

1. Return the value of this environment record's *MethodName*.

10.2.1.4 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 (10.3). A global environment provides the bindings for built-in globals (15.1), properties of the global object, and for all declarations that are not function code and that occur within *Script* productions.

A global environment record is logically a single record but it is specified as a composite encapsulating an object environment record and a declarative environment record. The object environment record has as its base object the global object of the associated Realm. This global object is also the value of the global environment record's *thisValue*. The object environment record component of a global environment record contains the bindings for all built-in globals (15.1) and all bindings introduced by a *FunctionDeclaration* or *VariableStatement* contained in global code. The bindings for all other ECMAScript declarations in global code are contained in the declarative environment record component of the global environment record.

Properties may be created directly on a global object. Hence, the object environment record component of a global environment record may contain both bindings created explicitly by *FunctionDeclaration* or *VariableStatement* declarations and binding created implicitly as properties of the global object. In order to identify which bindings were explicitly created using declarations, a global environment record maintains a list of the names bound using its CreateGlobalVarBindings and CreateGlobalFunctionBindings concrete methods.

Global environment records have the additional state components listed in Table 24 and the additional methods listed in Table 25.

Deleted: Method

Deleted: Regular

Deleted: Method

Deleted: HomeObject

Deleted: this environment record's *HomeObject*

Deleted:

Deleted: Otherwise, r

Deleted: value

Deleted: of

Deleted: Prototype

Deleted: property

Deleted: *HomeObject* value

Deleted: .

Deleted: is

Deleted: of a

Deleted: *Program*

Deleted: If such a function references **`super`**,

Deleted: *Program*

Deleted: . All ECMAScript *Program* productions that a processed iin a specific Realm (10.3) share the same global environment record..

Deleted: composit

Table 24 -- Components of Global Environment Records

Component	*Purpose*
ObjectEnvironment	A Object Environment Record whose base object is the global object. Contains global built-in bindings as well as bindings for *FunctionDeclaration* or *VariableStatement* declarations in global code for the associated Realm.
DeclarativeEnvironment	A Declarative Environment Record that contains bindings for all declarations in global for the associated Realm code except for *FunctionDeclaration* and *VariableStatement* declarations.
VarNames	A List containing the string names bound by *FunctionDeclaration* or *VariableStatement* declarations in global code for the associated Realm.

Table 25 — Additional Methods of Global Environment Records

Method	*Purpose*
GetThisBinding()	Return the value of this environment record's **`this`** binding.
HasVarDeclaration (N)	Determines if the argument identifier has a binding in this environment record that was created using a *VariableStatement* or a *FunctionDeclaration*.
HasLexicalDeclaration (N)	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*.
CanDeclareGlobalVar (N)	Determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument *N*.
CanDeclareGlobalFunction (N)	Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument *N*.
CreateGlobalVarBinding(N, D)	Used to create global **`var`** bindings in the ObjectEnvironmentComponent of the environment record. The binding will be a mutable binding. The corresponding global object property will have attribute values approate for a **`var`**. The String value *N* is the text of the bound name. *V* is the initial value of the binding If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted. This is logically equivalent to CreateMutableBinding but it allows var declarations to receive special treatment.
CreateGlobalFunctionBinding(N, V, D)	Used to create and initialise global **`function`** bindings in the ObjectEnvironmentComponent of the environment record. The binding will be a mutable binding. The corresponding global object property will have attribute values approate for a **`function`**.The String value *N* is the text of the bound name. If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted. This is logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows function declarations to receive special treatment.

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

10.2.1.4.1 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:

Deleted: o

Deleted: r

Deleted: :

Deleted: or

Deleted: z

Deleted: or

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

10.2.1.4.2 CreateMutableBinding (N, D)

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

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. Assert: *DclRec* does not already have a binding for *N*.
4. Create a mutable binding in *DclRec* for *N* and record that it is uninitialised. If *D* is **true** record that the newly created binding may be deleted by a subsequent DeleteBinding call.
5. Return NormalCompletion(empty)

10.2.1.4.3 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name *N* that is uninitialised. A binding must not already exist in this environment record for *N*.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create an immutable binding in *envRec* for *N* and record that it is uninitialised.

10.2.1.4.4 InitialiseBinding (N,V)

The concrete Environment Record method InitialiseBinding for global environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. An uninitialised binding for *N* must already exist.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Return the result of calling *DclRec's* InitialiseBinding concrete method with arguments *N* and *V*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Set the bound value for *N* in *envRec* to *V*.
   b. Record that the binding for *N* in *envRec* has been initialised.

10.2.1.4.5 SetMutableBinding (N,V,S)

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

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Return the result of calling the SetMutableBinding concrete method of *DclRec* with arguments *N*, *V*, and *S*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. Return the result of calling the SetMutableBinding concrete method of *ObjRec* with arguments *N*, *V*, and *S*.

Deleted: n
Deleted: n
Deleted: or
Deleted: and
Comment [AWB48]: This probably needs a D option argument, just like createMutableEnvironment
Deleted: InitializeBinding
Deleted: InitializeBinding
Deleted: n
Deleted: the
Deleted: InitializeBinding
Deleted: Declarative
Deleted: n
Deleted: n
Deleted: Declarative

10.2.1.4.6 GetBindingValue(N,S)

The concrete Environment Record method GetBindingValue for global environment records simply returns the value of its bound identifier whose name is the value of the argument *N*. If *S* is **true** and the binding is an uninitialised binding throw a **ReferenceError** exception. A property named *N* normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument *S*.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Return the result of calling the GetBindingValue concrete method of *DclRec* with arguments *N*, and *S*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. Return the result of calling the GetBindingValue concrete method of *ObjRec* with arguments *N*, and *S*.

10.2.1.4.7 DeleteBinding (N)

The concrete Environment Record method DeleteBinding for global environment records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Return the result of calling the DeleteBinding concrete method of *DclRec* with argument *N*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, then
   a. Let *status* be the result of calling the DeleteBinding concrete method of *ObjRec* with argument *N*.
   b. ReturnIfAbrupt(*status*).
   c. If *status* is **true**, then
      i. Let *varNames* be *envRec's* VarNames List.
      ii. If *N* is an element of *varNames*, then remove that element from the *varNames*.
   d. Return *status*.
6. Return **true**.

10.2.1.4.8 HasThisBinding ()

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

1. Return **true**.

10.2.1.4.9 HasSuperBinding ()

1. Return **false**.

10.2.1.4.10 WithBaseObject()

Global Environment Records always return **undefined** as their WithBaseObject.

1. Return **undefined**.

10.2.1.4.11 GetThisBinding ()

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

Deleted: n

Deleted: *V*,

Deleted: Declarative

Deleted: *V*,

Deleted: n

Deleted: n

Deleted: *Dcl*

10.2.1.4.12 HasVarDeclaration (N)

The concrete environment record method HasVarDeclaration for global environment records determines if the argument identifier has a binding in this record that was created using a *VariableStatement* or a *FunctionDeclaration*:

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *varDeclaredNames* be *envRec's* VarNames List.
3. If *varDeclaredNames* contains the value of *N*, return **true**.
4. Return **false**.

10.2.1.4.13 HasLexicalDeclaration (N)

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

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. Return the result of calling *DclRec's* HasBinding concrete method with argument *N*.

10.2.1.4.14 CanDeclareGlobalVar (N)

The concrete environment record method CanDeclareGlobalVar for global environment records determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument *N*. Redundent var declarations and var declarations for pre-existing global object properties are allowed.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, return **true**.
4. Let *bindings* be the binding object for *ObjRec*.
5. Let *extensible* be the result of IsExtensible(*bindings*).
6. Return *extensible*.

Deleted: <#>If the result of calling the [[GeIstExtensibl internal propertymethod of *bindings* is **true**, return **true**.¶

Deleted: **false**

10.2.1.4.15 CanDeclareGlobalFunction (N)

The concrete environment record method CanDeclareGlobalVar for global environment records determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument *N*.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Let *globalObject* be the binding object for *ObjRec*.
4. Let *extensible* be the result of IsExtensible(*globalObject*).
5. ReturnIfAbrupt(*extensible*).
6. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **false**, then return *extensible*.
7. Let *existingProp* be the result of calling the [[GetOwnProperty]] internal method of *globalObject* with argument *N*.
8. If *existingProp* is **undefined**, then return *extensible*.
9. If *existingProp*.[[Configurable]] is **true**, then return **true**.
10. If IsDataDescriptor(*existingProp*) is **true** and *existingProp* has attribute values {[[Writable]]: **true**, [[Enumerable]]: **true**}, then return **true**.
11. Return **false**.

Deleted: value

Deleted: calling of the [[GetIsExtensible]] internal propertymethod of

Comment [AWB1149]: Carry over from ES5, but perhaps unnecessary

10.2.1.4.16 CreateGlobalVarBinding (N, D)

The concrete Environment Record method CreateVarBinding for global environment records creates a mutable binding in the associated object environment record and records the bound name in the associated VarNames List. If a binding already exists, it is reused.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Assert: The result of calling *envRec's* CanDeclareGlobalVar concrete method with argument *N* is **true**.
4. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **false**, then
    a. Call the CreateMutableBinding concrete method of *ObjRec* with arguments *N* and *D*.
5. Let *varDeclaredNames* be *envRec's* VarNames List.
6. If *varDeclaredNames* does not contain the value of *N*, then
    a. Append *N* to *varDeclaredNames*.
7. Return.

10.2.1.4.17 CreateGlobalFunctionBinding (N, V, D)

The concrete Environment Record method CreateFunctionBinding for global environment records creates a mutable binding in the associated object environment record and records the bound name in the associated VarNames List. If a binding already exists, it is replaced.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Assert: The result of calling *envRec's* CanDeclareGlobalFunction concrete method with argument *N* is **true**.
4. Let *globalObject* be the binding object for *ObjRec*.
5. Let *existingProp* be the result of calling the [[GetOwnProperty]] internal method of *globalObject* with argument *N*.
6. If *existingProp* is **undefined** or *existingProp*.[[Configurable]] is **true**, then
    a. Call the [[DefineOwnProperty]] internal method of *globalObject* passing *N* and Property Descriptor {[[Value]]:*V*, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *D*} as arguments.
7. Else,
    a. Call the [[DefineOwnProperty]] internal method of *globalObject* passing *N* and Property Descriptor {[[Value]]:*V* } as arguments.
8. NOTE The assertion in step 3 means that the above [[DefineOwnProperty]] calls will never return **false** or an abrupt completion.
9. Let *varDeclaredNames* be *envRec's* VarNames List.
10. If *varDeclaredNames* does not contain the value of *N*, then
    a. Append *N* to *varDeclaredNames*.
11. Return.

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

10.2.2 Lexical Environment Operations

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

10.2.2.1 GetIdentifierReference (lex, name, strict)

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

1. If *lex* is the value **null**, then
    a. Return a value of type Reference whose base value is **undefined**, whose referenced name is *name*, and whose strict reference flag is *strict*.
2. Let *envRec* be *lex*'s environment record.
3. Let *exists* be the result of calling the HasBinding(*N*) concrete method of *envRec* passing *name* as the argument *N*.
4. If *exists* is `true`, then
    a. Return a value of type Reference whose base value is *envRec*, whose referenced name is *name*, and whose strict reference flag is *strict*.
5. Else
    a. Let *outer* be the value of *lex's* outer environment reference.
    b. Return the result of calling GetIdentifierReference passing *outer*, *name*, and *strict* as arguments.

Deleted: n
Deleted: s
Deleted: n
Deleted: ,
Deleted: ,
Deleted: , and false
Deleted: ,
Deleted: ,
Deleted: , and false
Comment [AWB1250]: TODO: need to reconsider this? May not be true if the global object can be a Proxy.
Deleted: s
Deleted: <#>¶
Deleted: n
Deleted: identifier
Deleted: mode
Deleted: mode

10.2.2.2 NewDeclarativeEnvironment (E)

When the abstract operation NewDeclarativeEnvironment is called with either a Lexical Environment or **null** as argument *E* the following steps are performed:

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

10.2.2.3 NewObjectEnvironment (O, E)

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

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

10.2.2.4 NewFunctionEnvironment (F, T)

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

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

10.3 10.3 Code Realms

Before it is evaluated, all ECMAScript code must be associated with a *Realm*. Conceptually, a realm consists of a set of intrinsic objects, an ECMAScript global environment, all of the ECMAScript code that is loaded within the scope of that global environment, a Loader object that can associate new ECMAScript code with the realm, and other associated state and resources.

A Realm is specified as a Record with the fields specified in Table 26:

Table 26 — Realm Record Fields

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

Deleted: Method

Deleted: Object

Deleted: t

Deleted: method

Deleted:

Deleted: <#>If *F's* [[ThisMode]] internal data proper is lexical, then ¶

Deleted: home

Comment [AWB851]: Other possible terms that hav been discussed are "Home" and "Island". We still need get final agreement on terminology.

Deleted: as

Deleted: n

Deleted: following

Deleted: t

The intrinsic objects associated with a code Realm include the well-known intrinsics listed in Table 11 and additional intrinsics specified by Table 27.

Table 27 — Additional Intrinsic Objects with Realm Specific Bindings

Intrinsic Name	*ECMAScript Language Association*
???	???
???	???

10.4 Execution Contexts

An *execution context* is a specification device that is used to track the runtime evaluation of code by an ECMAScript implementation. At any point in time, there is at most one execution context that is actually executing code. This is known as the *running* execution context. A stack is used to track execution contexts. The running execution context is always the top element of this stack. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context.

An execution context contains whatever implementation specific state is necessary to track the execution progress of its associated code. Each execution context has the state components listed in Table 28.

Table 28 — State Components for All Execution Contexts

Component	*Purpose*
code evaluation state	Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context.
Realm	The Realm from which associated code accesses ECMAScript resources.

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

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

Execution contexts for ECMAScript code have the additional state components listed in Table 29.

Table 29 —Additional State Components for ECMAScript Code Execution Contexts

Component	*Purpose*
LexicalEnvironment	Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.
VariableEnvironment	Identifies the Lexical Environment whose environment record holds bindings created by *VariableStatements* within this execution context.

The LexicalEnvironment and VariableEnvironment components of an execution context are always Lexical Environments. When an execution context is created its LexicalEnvironment and VariableEnvironment components initially have the same value. The value of the VariableEnvironment component never changes while the value of the LexicalEnvironment component may change during execution of code within an execution context.

Deleted: are

Deleted: Within this specification a reference such as %name% means the value with this intrinsic name in the [[intinsics]] record of the Realm of the running execution context.

Deleted: %Object%

Deleted: The initial value of the global object property named "Object".

Comment [AWB1352]: This is where we should list all the per realm intrinsics that don T have %names%

Deleted: %ObjectPrototype%

Deleted: The initial value of the "prototype" data property of the intrinsic %Object%.

Deleted: %ObjProto_toString%

Deleted: <#>10.2.3 The Global Environment¶
<#>The *global environment* is a unique Lexical Environment which is created before any ECMAScript code is executed. The global environment's Environment Record is an object environment record whose binding object is the global object

Deleted: n

Deleted: When control is transferred to

Deleted: to

Deleted: At any point in time, there is at most

Deleted: t

Deleted: and

Deleted: In addition, e

Deleted:

Deleted: Execution Context

Deleted: ECMASript

Deleted: PreviousContext

Deleted: within which the associated

Deleted: code is evaluated

Deleted: LexicalEnvironment

Deleted: ThisBinding

Deleted: The code evaluation state

Deleted: current

Deleted: current

Deleted: current

Deleted: ECMAScript

Deleted: current

Deleted: ECMAScript

Deleted: of

Deleted: current

Deleted: changes

Deleted: to

Deleted: current

Execution contexts representing the evaluation of generator objects have the additional state components listed in Table 30.

Table 30 -- Additional State Components for Generator Execution Contexts

Component	***Purpose***
Generator	The GeneratorObject that this execution context is evaluating.

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

Deleted: and "ThisBinding"

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

10.4.1 Identifier Resolution

Identifier resolution is the process of determining the binding of an *IdentifierName* using the LexicalEnvironment of the running execution context. During execution of ECMAScript code, Identifier Resolution is performed using the following algorithm:

Deleted: the syntactic production *PrimaryExpression* : *Identifier* is evaluated

1. Let *env* be the running execution context's LexicalEnvironment.
2. If the syntactic production that is being evaluated is contained in strict mode code, then let *strict* be **true**, else let *strict* be **false**.
3. Return the result of calling GetIdentifierReference abstract operation passing *env*, the StringValue of *IdentifierName*, and *strict* as arguments.

Deleted: a

Deleted: function

Deleted:

Deleted: v

Deleted: containing the same sequence of characters as

Deleted: *Identifier*

The result of evaluating an identifier is always a value of type Reference with its referenced name component equal to the *IdentifierName* String.

10.4.2 GetThisEnvironment

The abstract operation GetThisEnviroment finds the lexical environment that currently supplies the binding of the keyword **this**. GetThisEnviroment performs the following steps:

Deleted: It starts search with the LexicalEnvironment the current execution context.

1. Let *lex* be the running execution context's LexicalEnvironment.
2. Repeat
   a. Let *envRec* be *lex*'s environment record.
   b. Let *exists* be the result of calling the HasThisBinding concrete method of *envRec*.
   c. If *exists* is **true**, then return *envRec*.
   d. Let *outer* be the value of *lex's* outer environment reference.
   e. Let *lex* be *outer*.

Deleted: current

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.

Deleted: 4

10.4.3 This Resolution

The abstract operation ThisResolution is the process of determining the binding of the keyword **this** using the LexicalEnvironment of the running execution context. ThisResolution performs the following steps:

Deleted: current

1. Let *env* be the result of performing the GetThisEnvironment abstract operation.
2. Return the result of calling the GetThisBinding concrete method of *env*.

10.4.4 GetGlobalObject

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

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

10.5 Declaration Binding Instantiation

10.5.1 Global Declaration Instantiation

NOTE When an execution context is established for evaluating scripts, declarations are instantiated in the current global environment. Each global binding declared in the code is instantiated.

Global Declaration Instantiation is performed as follows using arguments *script*, *env*, and *deletableBindings*. *script* is the *ScriptBody* that for which the execution context is being established. *env* is the global environment record in which bindings are to be created. *deletableBindings* is **true** if the bindings that are created should be deletable.

1. Let *strict* be IsStrict of *script*.
2. Let *lexNames* be the LexicallyDeclaredNames of *script*.
3. Let *varNames* be the VarDeclaredNames of *script*.
4. For each *name* in *lexNames*, do
   a. If the result of calling *env's* HasVarDeclaration concrete method passing *name* as the argument is **true**, throw a **SyntaxError** exception.
   b. If the result of calling *env's* HasLexicalDeclaration concrete method passing *name* as the argument is **true**, throw a **SyntaxError** exception.
5. For each *name* in *varNames*, do
   a. If the result of calling *env's* HasLexicalDeclaration concrete method passing *name* as the argument is **true**, throw a **SyntaxError** exception.
6. Let *varDeclarations* be the VarScopedDeclarations of *script*.
7. Let *functionsToInitialise* be an empty List.
8. Let *declaredFunctionNames* be an empty List.
9. For each *d* in *varDeclarations*, in reverse list order do
   a. If *d* is a *FunctionDeclaration* then
      i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
      ii. Let *fn* be the sole element of the BoundNames of *d*.
      iii. If *fn* is not an element of *declaredFunctionNames*, then
         1. Let *fnDefinable* be the result of calling *env's* CanDeclareGlobalFunction concrete method passing *fn* as the argument.
         2. If *fnDefinable* is **false**, throw **TypeError** exception.
         3. Append *fn* to *declaredFunctionNames*.
         4. Append *d* to *functionsToInitialise*.
10. Let *declaredVarNames* be an empty List.
11. For each *d* in *varDeclarations*, do
    a. If *d* is a *VariableStatement* then
       i. For each String *vn* in the BoundNames of *d*, do
          1. If *vn* is not an element of *declaredFunctionNames*, then
             a Let *vnDefinable* be the result of calling *env's* CanDeclareGlobalVar concrete method passing *vn* and *deletableBindings* as the arguments.
             b If *vnDefinable* is **false**, throw **TypeError** exception.
             c If *vn* is not an element of *declaredVarNames*, then
                i. Append *vn* to *declaredVarNames*.
12. NOTE: No abnormal terminations occur after this algorithm step.
13. For each *FunctionDeclaration f* in *functionsToInitialise*, do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.

Deleted: P

Deleted: <#>10.4 Establishing an Execution Context¶
<#>Evaluation of global code or code using the eval function (15.1.2.1) establishes and enters a new execution context. Every invocation of an ECMAScript code function (13.2.1) also establishes and enters a new execution context, even if a function is calling itse recursively. Every return exits an execution context. A thrown exception may also exit one or more execution contexts.¶
<#>When control enters an execution context, the execution context's ThisBinding is set, its VariableEnvironment and initial LexicalEnvironment are defined, and declaration binding instantiation (10.5) i performed. The exact manner in which these actions occur depend on the type code being entered.¶
<#>10.4.1 Entering Global Code¶
<#>The following steps are performed when control enters the execution conte for global code:¶
<#>Initialise the execution context usin the global code as described in 10.4.1.1
<#>Perform Declaration Binding Instantiation as described in 10.5 using global code.¶
<#>10.4.1.1 Initial Global Execution Context¶
<#>The following steps are performed initialise a global execution context for ECMAScript code *C*:¶

Deleted: <#>10.5 Declaration Binding Instantiation¶

Comment [AWB53]: It may make sense to split eval code out into a separate subsection

Deleted: <#>Every execution context h an associated VariableEnvironment.

Deleted: Base Code

Deleted: Top-Level

Deleted: non-function code

Deleted: that is base code

Deleted: VariableEnvironment

Deleted: top-level

Deleted: level variable, constant, or function

Deleted: Top-level Declaration Instantiation base code is performed as follows: ¶

Deleted: **true** else let *strict* be **false**

Deleted: *code*

Deleted: *z*

Deleted: *z*

Deleted: *configurableBindings*

Deleted: *f*

Deleted: *z*

c. Call *env*'s CreateGlobalFunctionBinding concrete method passing *fn*, *fo*, and *deletableBindings* as the arguments.

14. For each String *vn* in *declaredVarNames*, in list order do

a. Call *env's* CreateGlobalVarBinding concrete method passing *vn* and *deletableBindings* as the argument.

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 initialised.

b. For each element *dn* of the BoundNames of *d* do

i. If IsConstantDeclaration of *d* is **true**, then

1. Call *env*'s CreateImmutableBinding concrete method passing *dn* as the argument.

ii. Else,

1. Call *env*'s CreateMutableBinding concrete method passing *dn* and **false** as the arguments.

c. If *d* is a *GeneratorDeclaration* production, then

i. Let *fn* be the sole element of the BoundNames of *d*.

ii. Let *fo* be the result of performing InstantiateFunctionObject for *d* with argument *env*.

iii. Call *env*'s SetMutableBinding concrete method passing *fn*, *fo*, and **false** as the arguments.

17. Return NormalCompletion(empty)

NOTE Early errors specified in 14.1 prevent name conflicts between function/var declarations and let/const/class/module declarations as well as redeclaration of let/const/class/module bindings for declaration contained within a single *Script*. However, such conflicts and redeclarations that span more than one *Script* are detected as runtime errors during Global Declaration Instantiation. If any such errors are detected, no bindings are instantiated for the script.

Unlike explicit var or function declarations, properties that are directly created on the global object result in global bindings that may be shadowed by let, const, class, and module declarations.

10.5.2 Module Declaration Instantiation

10.5.3 Function Declaration Instantiation

This version reflects the concensus as of the Sept. 2012 TC39 meeting. However, it now appears that the binding semantics of formal parameters is like to change again.

NOTE When an execution context is established for evaluating function code a new Declarative Environment Record is created and bindings for each formal parameter, and each function level variable, constant, or function declared in the function are instantiated in the environment record. Formal parameters and functions are initialised as part of this process. All other bindings are initialised during execution of the function code.

Function Declaration Instantiation is performed as follows using arguments *func*, *argumentsList*, and *env*. *func* is the function object that for which the execution context is being established. *env* is the declarative environment record in which bindings are to be created.

1. Let *code* be the value of the [[Code]] internal data property of *func*.
2. Let *strict* be the value of the [[Strict]] internal data property of *func*.
3. Let *formals* be the value of the [[FormalParameters]] internal data property of *func*.
4. Let *parameterNames* be the BoundNames of *formals*.
5. Let *varDeclarations* be the VarScopedDeclarations of *code*.
6. Let *functionsToInitialise* be an empty List.
7. If the value of the [[ThisMode]] internal data property of *func* is lexical, then

a. Let *argumentsObjectNeeded* be **false**.

8. Else, let *argumentsObjectNeeded* be **true**.

Deleted: Y

Deleted: *code*

Deleted: z

Comment [AWB54]: The exact details of declaration instantion for module code still need to be worked out.

Deleted: <#>If *code* is function code, then¶
<#>Let *func* be the function whose [[Call]] internal method initiated execution of *code*. Let *names* be the value of *func*'s [[FormalParameters]] internal property. ¶
<#>Let *argCount* be the number of elements in *args*.¶
<#>Let *n* be the number 0.¶
<#>For each String *argName* in *names*, in list order do¶
<#>Let *n* be the current value of *n* plus 1.¶
<#>If *n* is greater than *argCount*, let *v* be undefined otherwise let *v* be the value of the *n*'th element of *args*.¶
<#>Let *argAlreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *argName* as the argument.¶
<#>If argAlreadyDeclared is false, then¶
<#>call Call env's CreateMutableBinding concrete method passing argName as the argument.¶
<#>Call env's InitializeBinding concrete method passing argName, and undefined as the argume
<#>Call env's SetMutableBinding concrete meth passing argName, v, and strict as the arguments

Deleted: <#>For each FunctionDeclaration f in code, in source text order do¶
<#>Let fn be the Identifier in FunctionDeclaratio
<#>Let fo be the result of performing Instantiate Function Declaration forinstantiating FunctionDeclaration f as described in Clause 13.
<#>Let funcAlreadyDeclared be the result of calling env's HasBinding concrete method passi fn as the argument.¶
<#>If funcAlreadyDeclared is false, then ¶
<#>call Call env's CreateMutableVarBinding concrete method passing fn and configurableBindings as the arguments.¶
<#>Call env's InitializeBinding concrete method passing fn, and undefined as the arguments.¶

Deleted: Exrtended Code Top-Level

Deleted: <#>¶

Deleted: z

Deleted: z

Deleted: be

Deleted: *FormalParameterList*

Deleted: List

Deleted: *FormalParameterList*

Deleted: *z*

Comment [AWB1655]: Issue: should concise methods also not get an arguments object?

Deleted: L

Deleted: *Not*

Deleted: **false**

9. For each *d* in *varDeclarations*, in reverse list order do
   a. If *d* is a *FunctionDeclaration* then
      i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
      ii. Let *fn* be the sole element of the BoundNames of *d*.
      iii. If *fn* is "`arguments`", then let *argumentsObjectNeeded* be **false**.
      iv. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *fn* as the argument.
      v. If *alreadyDeclared* is **false**, then
         1. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *fn* as the argument.
         2. Assert: *status* is never an Abrupt Completion.
         3. Append *d* to *functionsToInitialise*.
10. For each String *paramName* in *parameterNames*, do
   a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *paramName* as the argument.
   b. NOTE Duplicate parameter names can only occur in non-strict functions. Parameter names that are the same as function declaration names do not get initialised to **undefined**.
   c. If *alreadyDeclared* is **false**, then
      i. If *paramName* is "`arguments`", then let *argumentsObjectNeeded* be **false**.
      ii. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *paramName* as the argument.
      iii. Assert: *status* is never an Abrupt Completion
      iv. Call *env's* InitialiseBinding concrete method passing *paramName*, and **undefined** as the arguments.
11. NOTE If there is a function declaration or formal parameter with the name "`arguments`" then an argument object is not created.
12. If *argumentsObjectNeeded* is **true**, then
   a. If *strict* is **true**, then
      i. Call *env's* CreateImmutableBinding concrete method passing the String "`arguments`" as the argument.
   b. Else,
      i. Call *env's* CreateMutableBinding concrete method passing the String "`arguments`" as the argument.
13. Let *varNames* be the VarDeclaredNames of *code*.
14. For each String *varName* in *varNames*, in list order do
   a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *varName* as the argument.
   b. NOTE A VarDeclaredNames is only instantiated and initialied here if it is not also the name of a formal parameter or a *FunctionDeclarations*.
   c. If *alreadyDeclared* is **false**, then
      i. Call *env's* CreateMutableBinding concrete method passing *varName* as the argument.
15. Let *lexDeclarations* be the LexicalDeclarations of *code*.
16. For each element *d* in *lexDeclarations* do
   a. NOTE A lexically declared name cannot be the same as a function declaration, formal parameter, or a var name. Lexically declarated names are only instantiated here but not initialised.
   b. For each element *dn* of the BoundNames of *d* do
      i. If IsConstantDeclaration of *d* is **true**, then
         1. Call *env's* CreateImmutableBinding concrete method passing *dn* as the argument.
      ii. Else,
         1. Call *env's* CreateMutableBinding concrete method passing *dn* and **false** as the arguments.
   c. If *d* is a *GeneratorDeclaration* production, then
      i. Append *d* to *functionsToInitialise*.
17. For each production *f* in *functionsToInitialise*, do
   a. Let *fn* be the sole element of the BoundNames of *f*.
   b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
   c. Call *env's* SetMutableBinding concrete method passing *fn*, *fo*, and **false** as the arguments.
18. NOTE Function declaration are initialised prior to parameter initialisation so that default value expressions may reference them. "`arguments`" is not initialised until after parameter initialization.

Deleted: *Not*
Deleted:
Deleted: **true**
Deleted: *z*
Deleted: *arg*
Deleted: in list order
Deleted: *arg*
Deleted: Normal
Deleted: code
Deleted: z
Deleted: *Not*
Deleted: **true**
Deleted: Call
Deleted: *arg*
Deleted: ReturnIfAbrupt(*status*).
Deleted: <#>If *strict* is **false**, then¶
Deleted: InitializeBinding
Deleted: *arg*
Deleted: *Not*
Deleted: **false**
Deleted:
Deleted: z
Deleted: *ize*
Deleted: FunctionDeclaration
Deleted: *ize*
Deleted: M
Deleted: it is not extended code.
Deleted: z

19. Let *ao* be the result of InstantiateArgumentsObject with argument *argumentsList*.
20. NOTE If *argumentsObjectNeeded* is **false** then the value of *ao* is not directly observable to ECMAScript code and need not actually exist. In that case, its use in the above steps is strictly as a device for specifying formal parameter initialisation semantics.
21. Let *formalStatus* be the result of performing Binding Initialisation for *formals* with *ao* and **undefined** as arguments.
22. ReturnIfAbrupt(*formalStatus*).
23. If *argumentsObjectNeeded* is **true**, then
    a. If *strict* is **true**, then
        i. Perform the abstract operation CompleteStrictArgumentsObject with argument *ao*.
    b. Else,
        i. Perform the abstract operation CompleteMappedArgumentsObject with arguments *ao*, *func*, *formals*, and *env*.
    c. Call *env*'s InitialiseBinding concrete method passing "`arguments`" and *ao* as arguments.
24. Return NormalCompletion(empty).

10.5.4 Block Declaration Instantiation

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

Block Declaration Instantiation is performed as follows using arguments *code* and *env*. *code* is the grammar production corresponding to the body of the block. *env* is the declarative environment record in which bindings are to be created.

1. Let *declarations* be the LexicalDeclarations of *code*.
2. Let *functionsToInitialise* be an empty List.
3. For each element *d* in *declarations* do
    a. For each element *dn* of the BoundNames of *d* do
        i. If IsConstantDeclaration of *d* is **true**, then
            1. Call *env*'s CreateImmutableBinding concrete method passing *dn* as the argument.
        ii. Else,
            1. Call *env*'s CreateMutableBinding concrete method passing *dn* and **false** as the arguments.
    b. If *d* is a *GeneratorDeclaration* production or a *FunctionDeclaration* production, then
        i. Append *d* to *functionsToInitialise*.
4. For each production *f* in *functionsToInitialise*, in list order do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
    c. Call *env*'s InitialiseBinding concrete method passing *fn*, and *fo* as the arguments.

10.5.5 Eval Declaration Instantiation

10.6 Arguments Object

When function code is evaluated, an arguments object is created unless (as specified in 10.5) the identifier **`arguments`** occurs as an *Identifier* in the function's *FormalParameters* or occurs as the *BindingIdentifier* of a *FunctionDeclaration* contained in the outermost *StatementList* of the function code.

The abstract operation InstantiateArgumentsObject called with an argument *args* performs the following steps:

1. Let *len* be the number of elements in *args*.
2. Let *obj* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
3. Call the [[DefineOwnProperty]] internal method on *obj* passing "`length`" and the Property Descriptor {[[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**} as arguments.
4. Let *indx* = *len* - 1.
5. Repeat while *indx* ≥ 0,
    a. Let *val* be the element of *args* at 0-origined list position *indx*.

Deleted: <#>Let *declarations* be the LexicalDeclaration of *code*.¶
<#>If *strict* is **true**, then¶
<#>Let *ao* be the result of CreateStrictArgumentsObje with argument *argumentsList*. ¶
Deleted: *Not*
Deleted: *argumentsAlreadyDeclared*
Deleted: **true**
Deleted: initialization
Deleted: <#>Let *hasArgumentsObject* be the Boolean
Deleted: *Not*
Deleted: *argumentsAlreadyDeclared*
Deleted: **false**
Deleted: *0*
Deleted: <#>Call *env*'s CreateImmutableBinding
Deleted: *0*
Deleted: <#>Call *env*'s CreateMutableBinding
Deleted: InitializeBinding
Deleted: <#>Let *varNames* be the VarDeclaredNames o
Deleted: .
Deleted: *hasArgumentsObject*
Comment [AWB756]: Jan 19 meeting nortes: Curre
Deleted: contained in extended code
Deleted: <#>Assert: *parameterNames* has no duplicate
Deleted: *ize*
Deleted: a *ConstDeclaration*
Deleted: *ize*
Deleted: FunctionDeclaration
Deleted: *declarations*
Deleted: *FunctionDeclaration*
Deleted: Instantiate Function Declaration
Deleted: instantiating *FunctionDeclaration*
Deleted: as described in Clause 13
Deleted: <#>Call *env's* CreateMutableBinding concr
Deleted: InitializeBinding
Deleted: ¶
Deleted: For each LetDeclaration and ConstDeclaration d
Deleted: control enters an execution context for
Deleted: *FormalParameterList*
Comment [AWB 257]: Additional modification to this
Deleted: *VariableDeclaration* or
Deleted: *Function*
Deleted: *D*
Deleted: <#>Add the [[NativeBrandBuiltinBrand]] inte
Deleted: ,
Deleted: , and **false**

b. Call the [[DefineOwnProperty]] internal method on *obj* passing ToString(*indx*) and the Property Descriptor {[[Value]]: *val*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**} as arguments.

c. Let *indx* = *indx* - 1

6. Return *obj*

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

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

The abstract operation *Complete*MappedArgumentsObject is called with object *obj*, object *func*, grammar production *formals*, and environment record *env*. *obj* must have been previously created by the abstract operation InstantiateArgumentsObject. The following steps are performed:

1. Let *len* be the result of Get(*obj*, `"length"`).
2. Let *mappedNames* be an empty List.
3. Let *numberOfNonRestFormals* be NumberOfParameters of *formals*.
4. Let *map* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
5. Let *indx* = *len* - 1.
6. Repeat while *indx* ≥ 0,
   a. If *indx* is less than the *numberOfNonRestFormals*, then
      i. Let *param* be getParameter of *formals* with argument *indx*.
      ii. If *param* is a *BindingIdentifier*, then
         1. Let *name* be the sole element of BoundNames of *param*.
         2. If *name* is not an element of *mappedNames*, then
            a Add *name* as an element of the list *mappedNames*.
            b Let *g* be the result of calling the MakeArgGetter abstract operation with arguments *name* and *env*.
            c Let *p* be the result of calling the MakeArgSetter abstract operation with arguments *name* and *env*.
            d Call the [[DefineOwnProperty]] internal method of *map* passing ToString(*indx*) and the Property Descriptor {[[Set]]: *p*, [[Get]]: *g*, [[Configurable]]: **true**} as arguments.
   b. Let *indx* = *indx* - 1
7. If *mappedNames* is not empty, then
   a. Set the [[ParameterMap]] internal data property of *obj* to *map*.
   b. Set the [[Get]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]] internal methods of *obj* to the definitions provided below.
8. Call the [[DefineOwnProperty]] internal method on *obj* passing `"callee"` and the Property Descriptor {[[Value]]: *func*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**} as arguments.
9. Return *obj*

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

1. Let *bodyText* be the result of concatenating the Strings "`return` ", *name*, and "`;`".
2. Let *body* be the result of parsing *bodyText* using *FunctionBody* as the goal symbol.
3. Let *parameters* be a *FormalParameters* **:** [empty] production.
4. Return the result of calling the abstract operation FunctionCreate using Normal as the *kind*, *parameters* as *FormalParameterList*, *body* for *FunctionBody*, *env* as *Scope*, and **true** for *Strict*.

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

1. Let *paramText* be the String *name* concatenated with the String "`_arg`".
2. Let *parameters* be the result of parsing *paramText* using *FormalParameters* as the goal symbol.
3. Let *bodyText* be the String "**<name> = <param>**`;`" with **<name>** replaced by the value of *name* and **<param>** replaced by the value of *paramText*.

Deleted: ,
Deleted: , and **false**
Deleted: CreateStrictArgumentsObject
Deleted: *list args*
Deleted: ion
Deleted: performs t
Deleted: <#>Let *len* be the number of elemen in *args*.¶ <#>Let *obj* be the result of the abstraction operation InstantiateArgumentsObject with argument *len*.¶ <#>Let *indx* = *len* - 1.¶
Deleted: or
Deleted: <#>Call the [[DefineOwnProperty]]
Deleted: *obj*
Deleted: Create
Deleted: List
Deleted: *names*
Deleted: ion
Deleted: , and argument list *args*
Deleted: performs the following steps
Deleted: number of elements in
Deleted: calling the [[
Deleted:]] internal method of
Deleted: with
Deleted: argument
Deleted: . *args*.
Deleted: <#>Let *obj* be the result of the
Deleted: >=
Deleted: <#>Let *val* be the element of *arg*
Deleted: number of elements in *names*
Deleted: *name*
Deleted: the element
Deleted: *names*
Deleted: at 0-origined list position
Deleted: ,
Deleted: , and **false**
Deleted: P
Deleted: ,
Deleted: p
Deleted: d
Deleted: , and **false**
Deleted: The abstract operation
Deleted: *List*
Deleted: creating a function object as describe
Deleted: no *FormalParameterList*
Deleted: *List*

4. Let *body* be the result of parsing *bodyText* using *FunctionBody* as the goal symbol.
5. Return the result of calling the abstract operation FunctionCreate using Normal as the *kind*, *parameters* as *FormalParameterList*, *body* for *FunctionBody*, *env* as *Scope*, and **true** for *Strict*.

The [[Get]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

1. Let *args* be the arguments object.
2. Let *map* be the value of the [[ParameterMap]] internal data property of the arguments object.
3. Let *isMapped* be the result of calling the [[GetOwnProperty]] internal method of *map* passing *P* as the argument.
4. If the value of *isMapped* is **undefined**, then
    a. Let *v* be the result of calling the default ordinary object [[Get]] internal method (8.12.3) on *args* passing *P* and *args* as the arguments.
    b. If *P* is `"caller"` and *v* is a strict mode Function object, throw a **TypeError** exception.
    c. Return *v*.
5. Else *map* contains a formal parameter mapping for *P*,
    a. Return the result of calling Get(*map*, *P*).

The [[GetOwnProperty]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

1. Let *desc* be the result of calling the default [[GetOwnProperty]] internal method (8.12.1) on the arguments object passing *P* as the argument.
2. If *desc* is **undefined** then return *desc*.
3. Let *map* be the value of the [[ParameterMap]] internal data property of the arguments object.
4. Let *isMapped* be the result of calling the [[GetOwnProperty]] internal method of *map* passing *P* as the argument.
5. If the value of *isMapped* is not **undefined**, then
    a. Set *desc*.[[Value]] to the result of calling Get(*map*, *P*).
6. Return *desc*.

The [[DefineOwnProperty]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* and Property Descriptor *Desc* performs the following steps:

1. Let *map* be the value of the [[ParameterMap]] internal data property of the arguments object.
2. Let *isMapped* be the result of calling the [[GetOwnProperty]] internal method of *map* passing *P* as the argument.
3. Let *allowed* be the result of calling the default [[DefineOwnProperty]] internal method (8.3.9) on the arguments object passing *P* and *Desc* as the arguments.
4. ReturnIfAbrupt(*allowed*).
5. If *allowed* is **false**, then return **false**.
6. If the value of *isMapped* is not **undefined**, then
    a. If IsAccessorDescriptor(*Desc*) is **true**, then
        i. Call the [[Delete]] internal method of *map* passing *P* as the argument.
    b. Else
        i. If *Desc*.[[Value]] is present, then
            1. Let *putStatus* be the result of Put(*map*, *P*, *Desc*.[[Value]], **false**).
            2. Assert: *putStatus* is **true** because formal parameters mapped by argument objects are always writable.
        ii. If *Desc*.[[Writable]] is present and its value is **false**, then
            1. Call the [[Delete]] internal method of *map* passing *P* as the argument.
7. Return **true**.

The [[Delete]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property key *P* performs the following steps:

1. Let *map* be the value of the [[ParameterMap]] internal data property of the arguments object.
2. Let *isMapped* be the result of calling the [[GetOwnProperty]] internal method of *map* passing *P* as the argument.

Deleted: creating a function object as described in 13.2
Deleted: a List containing the single String
Deleted: *param*
Comment [AWB1258]: TODO: keep reference up to date
Deleted: P
Deleted:
Deleted: *the arguments object*
Deleted: ,
Deleted: so
Deleted: the [[
Deleted:]] internal method of
Deleted: passing
Deleted: as the argument
Deleted: the [[
Deleted:]] internal method of
Deleted: passing
Deleted: as the argument
Deleted: ,
Deleted: , and Boolean flag *Throw*
Deleted: 12
Deleted: ,
Deleted: , and **false**
Deleted: ¶
If *Throw* is **true** then throw a **TypeError** exception, otherwise
Deleted: , and **false**
Deleted: s
Deleted: <#>Assert: the follow Put call will always succeed because formal parameters mapped by argument objects are always writable. ¶
Call
Deleted: the [[
Deleted:]] internal method of
Deleted:
Deleted: passing
Deleted: **and** ***Throw***
Deleted: as the arguments
Deleted: and **false**
Deleted: s
Deleted: name
Deleted: and Boolean flag *Throw*

3. Let *result* be the result of calling the default [[Delete]] internal method for ordinary objects (8.3.10) on the arguments object passing *P* as the argument.
4. If *result* is **true** and the value of *isMapped* is not **undefined**, then
   a. Call the [[Delete]] internal method of *map* passing *P* as the argument.
5. Return *result*.

NOTE 1 For non-strict mode functions the array index (defined in 15.4) data properties of an arguments object whose numeric name values are less than the number of formal parameters of the corresponding function object initially share their values with the corresponding argument bindings in the function's execution context. This means that changing the property changes the corresponding value of the argument binding and vice-versa. This correspondence is broken if such a property is deleted and then redefined or if the property is changed into an accessor property. For strict mode functions, the values of the arguments object's properties are simply a copy of the arguments passed to the function and there is no dynamic linkage between the property values and the formal parameter values.

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

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

11 Expressions

11.1 Primary Expressions

Syntax

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

CoverParenthesisedExpressionAndArrowParameterList **:**
  (*Expression* **)**
  (**)**
  (**...** *Identifier* **)**
  (*Expression* **,** **...** *Identifier* **)**

Supplemental Syntax

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

ParenthesisedExpression **:**
  (*Expression* **)**

Static Semantics

Static Semantics: CoveredParenthesisedExpression

Deleted: 12
Deleted: 7
Deleted: and *Throw*
Deleted: s
Deleted: , and **false**
Deleted: s
Deleted: named
Deleted: *ArrayLiteral*
Deleted: [Lexical goal *InputElementRegExp*
Deleted: *Quasi*
Deleted: *z*
Deleted: **(** *Expression* **)**
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *FormalParameterList*
Deleted: Static Semantics: Early Errors¶
PrimaryExpression **:** *CoverParenthesizedExpressionAndArrowParame List* ¶
<#>It is a Syntax Error if the lexical token sequence matched by *CoverParenthesizedExpressionAndArrowPara erList* cannot be parsed with no tokens left over using *ParenthesizedExpression* as the g symbol.¶
<#>All Early Errors rules for *ParenthesizedExpression* and its derived productions also apply to the CoveredParenthesizedExpression of *CoverParenthesizedExpressionAndArrowPara erList*.¶
¶
Deleted: z

CoverParenthesisedExpressionAndArrowParameterList **:** **(** *Expression* **)**

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

Deleted: z

Deleted: z

Deleted: z

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

PrimaryExpression **:** *Identifier*

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

PrimaryExpression **:** *CoverParenthesisedExpressionAndArrowParameterList*

Deleted: z

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

Deleted: z

Deleted: z

11.1.1 The `this` Keyword

Runtime Semantics: Evaluation

PrimaryExpression **:** **`this`**

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

Deleted: <#>Let *env* be the result of performing the GetThisEnvironment abstract operation.¶

Deleted: GetThisBinding

Deleted: concrete method of *env*

Deleted: <#>The this keyword evaluates to Ret... the value of the ThisBinding of the current execution context.¶

11.1.2 Identifier Reference

Runtime Semantics: Evaluation

Comment [AWB859]: It may make sense to define some of the static semantic rules related static name resolution

PrimaryExpression **:** *Identifier*

1. Let *ref* be the result of performing Identifier Resolution as specified in 10.4.1 using the *IdentifierName* corresponding to *Identifier*.
2. Return *ref*.

Deleted: 3

NOTE: The result of evaluating an *Identifier* is always a value of type Reference.

Deleted: An *Identifier* is evaluated by performing Identifier Resolution as specified in 10.3.1 using the *IdentifierName* corresponding to *Identifier*.

11.1.3 Literals

Deleted: Reference

Syntax

Literal **:**
  NullLiteral
  ValueLiteral

ValueLiteral **:**
BooleanLiteral
NumericLiteral
StringLiteral

Deleted: *RegularExpressionLiteral*

Runtime Semantics

Runtime Semantics: Evaluation

Literal **:** *NullLiteral*

1. Return **null**.

ValueLiteral **:** *BooleanLiteral*

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

ValueLiteral **:** *NumericLiteral*

1. Return the number whose value is MV of *NumericLiteral* as defined in 7.8.3.

ValueLiteral **:** *StringLiteral*

1. Return the string whose elements are the SV of *StringLiteral* as defined in 7.8.4.

Deleted: character

Deleted: A *Literal* is evaluated as described 7.8.

11.1.4 Array Initialiser

Syntax

ArrayInitialiser **:**
ArrayLiteral
ArrayComprehension

Deleted: *SealedArrayInitialiser* **:**¶
ArrayInitialiser
ArrayInitialiser¶

11.1.4.1 Array Literal

NOTE An *ArrayLiteral* is an expression describing the initialisation of an Array object, using a list, of zero or more expressions, each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initialiser is evaluated.

Deleted: *array initialiser*

Deleted: written in a form of a literal. It is

Deleted: ,

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an *AssignmentExpression* (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.

Syntax

ArrayLiteral **:**
[$Elision_{opt}$ **]**
[*ElementList* **]**
[*ElementList* **,** $Elision_{opt}$ **]**

ElementList **:**
$Elision_{opt}$ *AssignmentExpression*
$Elision_{opt}$ *SpreadElement*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*
ElementList **,** $Elision_{opt}$ *SpreadElement*

Deleted: … *AssignmentExpression*

Deleted: … *AssignmentExpression*

Elision **:**

,

Elision **,**

SpreadElement **:**

... *AssignmentExpression*

Static Semantics

Static Semantics: Elision Width

Elision **: ,**

1. Return the numeric value 1.

Elision **:** *Elision* **,**

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

Runtime Semantics

Runtime Semantics: Array Accumulation

With parameters *array* and *nextIndex*.

ElementList **:** *Elision*$_{opt}$ *AssignmentExpression*

1. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *initResult* be the result of evaluating *AssignmentExpression*.
3. Let *initValue* be GetValue(*initResult*).
4. ReturnIfAbrupt(*initValue*).
5. Let *created* be the result of calling the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32(*nextIndex*+*padding*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
6. Assert: *created* is **true**.
7. Return *nextIndex*+*padding*+1.

ElementList **:** *Elision*$_{opt}$ *SpreadElement*

1. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing Array Accumulation for *SpreadElement* with arguments *array* and *nextIndex*+*padding*.

ElementList **:** *ElementList* **,** *Elision*$_{opt}$ *AssignmentExpression*

1. Let *postIndex* be the result of performing Array Accumulation for *ElementList* with arguments *array* and *nextIndex*.
2. ReturnIfAbrupt(*postIndex*).
3. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Let *initResult* be the result of evaluating *AssignmentExpression*.
5. Let *initValue* be GetValue(*initResult*).
6. ReturnIfAbrupt(*initValue*).
7. Let *created* be the result of calling the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32(*postIndex*+*padding*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. Assert: *created* is **true**.
9. Return *postIndex*+*padding*+1.

ElementList **:** *ElementList* **,** *Elision*$_{opt}$ *SpreadElement*

Deleted: ...

Deleted: *ArrayComprehension* **:**¶
Expression ComprehensionForList
Expression ComprehensionForList `if` (*Expression*)
ComprehensionForList **:**¶
ComprehensionFor
ComprehensionForList ComprehensionFor¶
ComprehensionFor **:**¶
`for` (*LeftHandSideExpression* `of` *Expression*)¶

Deleted: 11.1.4.1

Deleted: Static Semantics: Early Errors¶
Early Errors¶
The static semantics of the production SpreadElementElementList **:** *Elision*$_{opt}$ ... *AssignmentExpression* are:¶
<#>It is a Syntax Error if the scourcesource code parsed with this production is not extended code.¶

Deleted: *ElementList* **:** *ElementList* **,** *Elision*$_{opt}$ ... *AssignmentExpression* ¶
<#>It is a Syntax Error if the source code parsed wi this production is not extended code.¶

Deleted: *Elision* **:** [empty] ¶
<#>Return the numeric value 0.¶

Deleted: 11.1.4.2

Deleted: opt

Deleted: If *initValue* is an abrupt completion, return *initValue*

Deleted: C

Deleted: all

Deleted: ,

Deleted: , and **false**

Formatted: Font: Italic, Complex Script Font: Italic

Deleted:

Deleted: *the above call to [[DefineOwnProperty]] will never return* ***false*** *or an abrupt completion value*

Formatted: Font: Italic, Complex Script Font: Italic

Deleted: opt

Deleted: If *postIndex* is an abrupt completion, return *postIndex*

Deleted: opt

Deleted: If *initValue* is an abrupt completion, return *initValue*

Deleted: Call

Deleted: (

Deleted: , and **false**

Deleted: the above call to [[DefineOwnProperty]] will never return **false** or an abrupt completion value

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

SpreadElement **:** `...` *AssignmentExpression*

1. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
2. Let *spreadValue* be GetValue(*spreadRef*).
3. Let *spreadObj* be ToObject(*spreadValue*).
4. ReturnIfAbrupt(*spreadObj*).
5. Let *lenVal* be the result of calling Get(*spreadObj*, `"length"`).
6. Let *spreadLen* be ToUint32(*lenVal*).
7. ReturnIfAbrupt(*spreadLen*).
8. Let *n*=0;
9. Repeat, while *n* < *spreadLen*
   a. Let *exists* be the result of HasProperty(*spreadObj*, ToString(*n*)).
   b. ReturnIfAbrupt(*exists*).
   c. If *exists* is **true** then,
      i. Let *v* be the result of calling the [[Get]] internal method of *spreadObj* passing ToString(*n*) as the argument.
      ii. ReturnIfAbrupt(*v*).
      iii. Let *created* be the result of calling the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32(*nextIndex*)) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
      iv. Assert: *created* is **true**.
   d. Let *n* = *n*+1.
   e. Let *nextIndex* = *nextIndex* +1.
10. Return *nextIndex*.

NOTE [[DefineOwnProperty]] is used to ensure that own properties are defined for the array even if the standard built-in Array prototype object has been modified in a manner that would preclude the creation of new own properties using [[Set]].

Runtime Semantics: Evaluation

ArrayLiteral **:** `[` *Elision*$_{opt}$ `]`

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *pad* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
3. Call Put(*array*, `"length"`, *pad*, **false**).
4. Return *array*.

ArrayLiteral **:** `[` *ElementList* `]`

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *len* be the result of performing Array Accumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*len*).
4. Call Put(*array*, `"length"`, *len*, **false**).
5. Return *array*.

ArrayLiteral **:** `[` *ElementList* `,` *Elision*$_{opt}$ `]`

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *len* be the result of performing Array Accumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*len*).
4. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.

Deleted: If *postIndex* is an abrupt completion return *postIndex*
Deleted: Let *pad* be the result of evaluating *Elision*$_{opt}$; if not present, use the numeric value zero.
Deleted: ...
Comment [AW60]: Note that the value the spread operator is applied to is coerced to an Object.
Deleted: If *spreadObj* is an abrupt completion return *spreadObj*
Deleted: the [[…et(]] internal method of
Deleted: calling the [[HasProperty]] internal method of …*preadObj* with
Deleted: Call
Comment [AW61]: Note that indices wrap. For example consider:
[,,,,,, …{ 4294967293: "x", length: Math.pow(2,32)-2}]
Deleted: ,…and Property Descriptor
Deleted: the above call to [[DefineOwnProperty]] will never return fals or an abrupt completion value
Deleted: Put
Deleted: P
Deleted: The production …*rrayLiteral* **:**
Deleted: ion
Deleted: (15.4)
Deleted: the result of creating a new object as by the expression `new Array()` where `Arr` is the standard built-in constructor with that nam
Deleted: result of evaluating
Deleted: opt
Deleted: ; if not present, use the numeric valu zero
Deleted: the [[…ut]] internal method of… …r
Deleted: The production …*rrayLiteral* **:**
Deleted: ion
Deleted: (15.4)
Deleted: the result of creating a new object as by the expression `new Array()` where `Arr`
Deleted: Return the…et *len* be the result of
Deleted: If *len* is an abrupt completion, return
Deleted: the [[…ut]] internal method of… …r
Deleted: The production …*rrayLiteral* **:**
Deleted: ion
Deleted: (15.4)
Deleted: the result of creating a new object as
Deleted: <#>Let *array* be the result of
Deleted: If *len* is an abrupt completion, return
Deleted: Let *pad* be the Elision Width of resul

5. Call Put(*array*, **"length"**, ToUint32(*padding*+*len*), **false**).
6. Return *array*.

11.1.4.2 Array Comprehension

Syntax

ArrayComprehension **:**
  [*Comprehension* **]**

Comprehension **:**
  ComprehensionFor ComprehensionQualifierTail

ComprehensionQualifierTail **:**
  AssignmentExpression
  ComprehensionQualifier ComprehensionQualifierTail

ComprehensionQualifier **:**
  ComprehensionFor
  ComprehensionIf

ComprehensionFor **:**
  for (*ForBinding* **of** *AssignmentExpression* **)**

ComprehensionIf **:**
  if (*AssignmentExpression* **)**

ForBinding **:**
  BindingIdentifier
  BindingPattern

Static Semantics

Static Semantics: Early Errors

ComprehensionFor **: for (** *ForBinding* **of** *AssignmentExpression* **)**

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

Runtime Semantics

Runtime Semantics: Binding Initialisation

With arguments *value* and *environment*.

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

ForBinding **:** *BindingPattern*

1. Let *obj* be ToObject(*value*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of performing Binding Initialisation for *BindingPattern* passing *obj* and *environment* as the arguments.

Runtime Semantics: ComprehensionEvaluation

With argument *accumulator*.

Deleted: <#>Let *len* be the result of calling the [[Get]] internal method of *array* with argument **"length"**.

Deleted: the [[

Deleted:]] internal method of

Deleted:

Deleted: with arguments

Deleted: and

Deleted: **[** *AssignmentExpression ComprehensionForList* **]**

Deleted: *Assignment Expression*

Deleted: *ComprehensionForList* **if (** *Expression*

Deleted: **)**

Deleted:

Deleted: *Comprehension* **:**
ComprehensionQualification AssignmentExpression
ComprehensionQualification **:**
ComprehensionFor ComprehensionQualifierList$_{opt}$

Deleted: *ComprehensionForQualifierList* **:**
ComprehensionForQualifier

ComprehensionForQualifierList ComprehensionForQualifier

Deleted: **(**

Deleted: *LeftHandSideExpression*

Deleted: *Expression*

Deleted: **)**

Deleted: **Static Semantics**
Static Semantics: Bound Names
ForBinding : BindingIdentifier
<#>Return the Bound Names of BindingIdentifier
ForBinding : BindingPattern
<#>Return the Bound Names of BindingPattern.

Deleted: z

Deleted: **[**

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

Comprehension **:** *ComprehensionFor ComprehensionQualifierTail*

1. Return the result of performing QualifierEvaluation for *ComprehensionFor* with arguments *ComprehensionQualifierTail* and *accumulator*.

ComprehensionQualifierTail **:** *ComprehensionQualifier ComprehensionQualifierTail*

1. Return the result of performing QualifierEvaluation for *ComprehensionQualifier* with arguments *ComprehensionQualifierTail* and *accumulator*.

ComprehensionQualifierTail **:** *AssignmentExpression*

1. Let *valueRef* be the result of evaluating *AssignmentExpression*.
2. Let *value* be GetValue(*valueRef*).
3. ReturnIfAbrupt(*value*).
4. If *accumulator* is not **undefined**, then
    a. Assert: this is part of an array comprehension.
    b. Assert: *accumulator* is an exotic array object so access to its **`length`** property should never fail.
    c. Let *len* be the result of Get(*accumulator*, **`"length"`**).
    d. If $len \geq 2^{32}$-1, then throw a **RangeError** exception.
    e. Let *putStatus* be the result of Put(*O*, ToString(*len*), *value*, **true**).
    f. ReturnIfAbrupt(*putStatus*).
    g. Increase *len* by 1.
    h. Let *putStatus* be the result of Put(*O*, **`"length"`**, *len*, **true**).
    i. ReturnIfAbrupt(*putStatus*).
    j. Return NormalCompletion(**undefined**).
5. Assert: *accumulator* is **undefined**, so this is part of a generator comprehension.
6. Let *yieldStatus* be the result of GeneratorYield(CreateItrResultObject(*value*, **false**)).
7. ReturnIfAbrupt(*yieldStatus*).
8. Return NormalCompletion(**undefined**).

Runtime Semantics: QualifierEvaluation

With arguments *tail* and *accumulator*.

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

ComprehensionFor **:** **`for`** **(***ForBinding* **`of`** *AssignmentExpression* **)**

1. Let *exprRef* be the result of evaluating *AssignmentExpression*.
2. Let *exprValue* be GetValue(*exprRef*).
3. Let *obj* be ToObject(*exprValue*).
4. ReturnIfAbrupt(*obj*).
5. Let *iterator* be the result of performing Invoke with arguments *obj*, @@iterator, and an empty List.
6. Let *keys* be ToObject(*iterator*).
7. ReturnIfAbrupt(*keys*).
8. Let *oldEnv* be the running execution context's LexicalEnvironment.
9. Let *noArgs* be an empty List.
10. Repeat
    a. Let *nextResult* be the result of Invoke(*keys*, **`"next"`**).
    b. ReturnIfAbrupt(*nextResult*).
    c. If Type(*nextResult*) is not Object, then throw a **TypeError** exception.
    d. Let *done* be IteratorComplete(*nextResult*).

Deleted: *e*

Deleted: *e*

e. ReturnIfAbrupt(*done*).
f. If *done* is **true**, then return **true**.
g. Let *nextValue* be IteratorValue(*nextResult*);
h. ReturnIfAbrupt(*nextValue*).
i. Let *forEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
j. For each element *name* of the BoundNames of *ForBinding* do
   i. Call *forEnv*'s CreateMutableBinding concrete method with argument *name*.
k. Let *status* be the result of performing Binding Initialisation for *ForBinding* passing *value* and *environment* as the arguments.
l. ReturnIfAbrupt(*status*).
m. Set the running execution context's LexicalEnvironment to *forEnv*.
n. Let *continue* be the result of performing ComprehensionEvaluation for *tail* with argument *accumulator*.
o. Set the running execution context's LexicalEnvironment to *oldEnv*.
p. ReturnIfAbrupt(*continue*).

ComprehensionFor **: if (*AssignmentExpression*)**

1. Let *valueRef* be the result of evaluating *AssignmentExpression*.
2. Let *value* be GetValue(*valueRef*).
3. Let *boolValue* be ToBoolean(*value*).
4. ReturnIfAbrupt(*boolValue*).
5. If *exprValue* is **true**, then
   a. Return the result of performing ComprehensionEvaluation for *tail* with argument *accumulator*.
6. Else,
   a. Return NormalCompletion(**undefined**).

Runtime Semantics: Evaluation

ArrayComprehension **: [** *Comprehension* **]**

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *status* be the result of performing ComprehensionEvaluation for *Compression* with argument *array*.
3. ReturnIfAbrupt(*status*).
4. Return *array*.

11.1.5 Object Initialiser

NOTE 1 An object initialiser is an expression describing the initialisation of an Object, written in a form resembling a literal. It is a list of zero or more pairs of property names and associated values, enclosed in curly braces. The values need not be literals; they are evaluated each time the object initialiser is evaluated.

Syntax

ObjectLiteral **:**
  { }
  { *PropertyDefinitionList* **}**
  { *PropertyDefinitionList* **, }**

PropertyDefinitionList **:**
  PropertyDefinition
  PropertyDefinitionList **,** *PropertyDefinition*

PropertyDefinition **:**
  IdentifierName
  CoverInitialisedName
  PropertyName **:** *AssignmentExpression*
  MethodDefinition

Deleted: Runtime Semantics: Binding Initialisation
With arguments *value* and *environment*.¶
NOTE **undefined** is passed for *environment* to indicate that a PutValue operation should be used to assign the initialisation value. This is the case for **var** statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialized prior evaluation of its initializer. ¶
ForBinding **:** *BindingIdentifier* ¶
<#>Perform Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.¶
ForBinding: *BindingPattern*¶
<#>Perform Binding Initialisation for *BindingPattern* passing *value* and *env* as the arguments.¶

Deleted: <#>*ArrayComprehensionLiteral* :
[*ElementList*] ¶
<#>Let array be the result of the abstraction operation ArrayCreate (15.4) with argument 0.¶
<#>Let len be result of performing Array Accumulation for ElementList with arguments array and 0.¶
<#>ReturnIfAbrupt(len)If len is an abrupt completion, return len.¶
<#>Call the [[Put]] internal method of array with arguments "length", len, and false.¶
<#>Return array.¶

Deleted: <#>The production *ElementList* : *Elision*$_{opt}$ *AssignmentExpression* is evaluated as follows:¶
<#>Let array be the result of creating a new obj as if by the expression new Array() where Array is the standard built-in constructor with that name.¶
<#>Let firstIndex be the result of evaluating Elision; if not present, use the numeric value zer
<#>Let initResult be the result of evaluating AssignmentExpression.¶
<#>Let initValue be GetValue(initResult).¶
<#>Call the [[DefineOwnProperty]] internal meth of array with arguments ToString(firstIndex), the Property Descriptor { [[Value]]: initValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}, and false.¶
<#>Return array.¶
<#>The production *ElementList* : *Elision*$_{opt}$ ... *AssignmentExpression* is evaluated as follows:¶
<#>Let array be the result of creating a new obj as if by the expression new Array() where Array is the standard built-in constructor with that name.¶
<#>Let index be the result of evaluating Elision; not present, use the numeric value zero.¶
<#>Let spreadRef be the result of evaluating AssignmentExpression.¶
<#>Let spreadValue be GetValue(spreadRef).¶
<#>Let spreadObj be ToObject(spreadValue).¶

Deleted: *PropertyNameAndValueList*
Deleted: *PropertyNameAndValueList*
Deleted: *PropertyNameAndValueList*
Deleted: *PropertyAssignment*
Deleted: *PropertyNameAndValueList*
Deleted: *PropertyAssignment*
Deleted: *PropertyAssignment*

PropertyName **:**
  LiteralPropertyName
  ComputedPropertyName

LiteralPropertyName **:**
  IdentifierName
  StringLiteral
  NumericLiteral

ComputedPropertyName **:**
  [*AssignmentExpression* **]**

CoverInitialisedName **:**
  IdentifierName Initialiser

Initialiser **:**
  = *AssignmentExpression*

NOTE 2 *MethodDefinition* is defined in 13.3.

NOTE 3 In certain contexts, *ObjectLiteral* is used as a cover grammar for a more restricted secondary grammar. The *CoverInitialisedName* production is necessary to fully cover these secondary grammars. However, use of this production results in an early Syntax Error in normal contexts where an actual *ObjectLiteral* is expected.

Static Semantics

Static Semantics: Early Errors

In addition to describing an actual object initialiser the *ObjectLiteral* productions are also used as a cover grammar for *ObjectAssignmentPattern* (11.13.1). When *ObjectLiteral* appears in a context where *ObjectAssignmentPattern* is required, the following Early Error rules are **not** applied.

ObjectLiteral **:** **{** *PropertyDefinitionList* **}**
and
ObjectLiteral **:** **{** *PropertyDefinitionList* **,** **}**

- It is a Syntax Error if PropertyNameList of *PropertyDefinitionList* contains any duplicate entries, unless one of the following conditions are true for each duplicate entry:
  1. The source code corresponding to *PropertyDefinitionList* is not strict code and all occurrences in the list of the duplicated entry were obtained from productions of the form *PropertyDefinition* **:** *PropertyName* **:** *AssignmentExpression*.
  2. The duplicated entry occurs exactly twice in the list and one occurrence was obtained from a `get` accessor *MethodDefinition* and the other occurrence was obtained from a `set` accessor *MethodDefinition*.

PropertyDefinition **:** *MethodDefinition*

- It is a Syntax Error if ReferencesSuper of *MethodDefinition* is **true**.

PropertyDefinition **:** *IdentifierName*

- It is a Syntax Error if *IdentifierName* is a *ReservedWord*.

PropertyDefinition **:** *CoverInitialisedName*

- Always throw a Syntax Error if this production is present

Deleted: *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
get *PropertyName* **(** **)** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterLi*
) **{** *FunctionBody* **}**

Deleted: 1

Deleted: 2

Deleted: <#>PropertySetParameterLis
<#>BindingIdentifier
BindingPattern ¶

Deleted: <#>Identifier¶
<#>Semantics¶

Deleted: e

Deleted: *PropertyNameAndValueList* **:**
PropertyNameAndValueList **,** *PropertyAssignme*

Deleted: a

Deleted: *property definition*

Deleted: property definition

Deleted: *i*

Comment [AWB862]: The currently prevailing position in TC39 is that use of supe should not be allowed in object literals. This restriction is arbitrary in the sense that the runtime semantics would work.

Deleted: *i*

NOTE This production exists so that *ObjectLiteral* can serve as a cover grammar for *ObjectAssignmentPattern* (11.13.1). It cannot occur in an actual object initialiser.

Static Semantics: Contains

With parameter *symbol*.

PropertyDefinition **:** *MethodDefinition*

1. If *symbol* is MethodDefinition, return **true**.
2. Return **false**.

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

LiteralPropertyName **:** *IdentifierName*

1. If *symbol* is a *ReservedWord*, return **false**.
2. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
3. Return **false**.

Static Semantics: IsValidSimpleAssignmentTarget

PrimaryExpression **:** *Literal*

1. Return **false**.

Static Semantics: PropName

PropertyDefinition **:** *IdentifierName*

1. Return StringValue of *IdentifierName*.

PropertyDefinition **:** *PropertyName* **:** *AssignmentExpression*

1. Return PropName of *PropertyName*.

LiteralPropertyName **:** *StringLiteral*

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

LiteralPropertyName **:** *NumericLiteral*

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

ComputedPropertyName **:** **[** *AssignmentExpression* **]**

1. Return empty.

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*

Deleted:

Deleted: <#>It is a Syntax Error if this production i contained in strict code, *PropertyAssignment* is the production *PropertyAssignment* **:** *PropertyName* **:** *AssignmentExpression*, and PropertyDefinitionList(PropName of *PropertyAssignmen* of *PropertyNameAndValueList* is not the empty List.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production *PropertyAssignment* **:** *IdentifierName* and PropertyDefinitionList(PropName of *PropertyAssignmen* of *PropertyNameAndValueList* is not the empty List.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production *PropertyAssignment* **:** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}** and PropertyDefinitionList(PropName of *PropertyAssignmen* of *PropertyNameAndValueList* is not the empty List.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production *PropertyAssignment* **:** * *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}** and PropertyDefinitionList(PropName of *PropertyAssignmen* of *PropertyNameAndValueList* is not the empty List.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production
PropertyAssignment **:** **get** *PropertyName* **(** **)** **{** *FunctionBody* **}**
and PropertyDefinitionList (PropName of *PropertyAssignment*) of *PropertyNameAndValueList* includes a production of the form *PropertyAssignment* *PropertyName* **:** *AssignmentExpression*.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production
PropertyAssignment **:** **set** *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**
and PropertyDefinitionList (PropName of *PropertyAssignment*) of *PropertyNameAndValueList* includes a production of the form *PropertyAssignment* *PropertyName* **:** *AssignmentExpression*.¶
<#>It is a Syntax Error if *PropertyAssignment* is the production
PropertyAssignment **:** **get** *PropertyName* **(** **)**

Deleted: *PropertyDefinition* **:** *MethodDefinition* ¶

Deleted:

Deleted:

Deleted: NOTE Static semantic rules that depend up substructure generally do not look into function definiti

Deleted: Static Semantics: ExpectedArgumentCou
PropertySetParameterList **:** *BindingIdentifier* ¶
<#>Return 1.¶

Deleted: *PropertyAssignment*

Deleted:

Deleted: *PropertyAssignment*

Deleted: *PropertyDefinition* **:** *MethodDefinition* ¶
<#>Return PropName of *MethodDefinition*.¶

Deleted: *PropertyAssignment* **:** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}** ¶
<#>Return PropName of *PropertyName*.¶

Deleted: *PropertyName* **:** *IdentifierName* ¶
<#>Return PropName(of *IdentifierName*).¶

Deleted: the

Deleted: s

Comment [AWB1063]: Issue: static semantic rules probably should call ToString (a runtime operation).

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*.

Deleted:

Runtime Semantics

Deleted: 11.1.5.2

Runtime Semantics: Evaluation

ObjectLiteral : **{ }**

Deleted: The production

Deleted: is evaluated as follows:

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

Deleted: a new object created as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name

ObjectLiteral :
 { *PropertyDefinitionList* **}**
 { *PropertyDefinitionList* **, }**

Deleted: *PropertyNameAndValueList*

Deleted: *PropertyNameAndValueList*

Deleted: *ObjectLiteral : { PropertyNameAndValueList } and ObjectLiteral : { PropertyNameAndValueList , } are evaluated as follows:*

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

Deleted: ion

Deleted: (15.2)

Deleted: creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name

Deleted: P

Deleted: *success*

Deleted: D

Deleted: *PropertyNameAndValueList*

Deleted: If *successstatus* is an abrupt completi return *successstatus*

Deleted: the result of evaluating *PropertyNameAndValueList*

PropertyDefinition **:** *IdentifierName*

1. Return StringValue of *IdentifierName*.

PropertyDefinition **:** *PropertyName* **:** *AssignmentExpression*

1. Return the result of evaluating *PropertyName*.

LiteralPropertyName: *IdentifierName*

1. Return StringValue of *IdentifierName*.

LiteralPropertyName **:** *StringLiteral*

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

LiteralPropertyName **:** *NumericLiteral*

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

ComputedPropertyName **:** **[** *AssignmentExpression* **]**

1. Let *exprValue* be the result of evaluating *AssignmentExpression*.
2. Let *propName* be GetValue(*exprValue*).
3. ReturnIfAbrupt(*propName*).
4. If *propName* is not an exotic Symbol Object, then
   a. Throw a **TypeError** exception.
5. Return *propName*.

Comment [AWB1664]: Issue, alternatively we could ToPropertyKey it if we want to allow non-symbol keys to be used as literal propert names.

Runtime Semantics: Property Definition Evaluation

With parameter *object* and optional parameter *functionPrototype*.

Deleted: .

PropertyDefinitionList **:** *PropertyDefinitionList* **,** *PropertyDefinition*

1. Let *status* be the result of performing Property Definition Evaluation of *PropertyDefinitionList* with argument *object*.
2. ReturnIfAbrupt(*status*).
3. Return the result of performing Property Definition Evaluation of *PropertyDefinition* with argument *object*.

PropertyDefinition **:** *IdentifierName*

1. Let *propName* be StringValue of *IdentifierName*.
2. Let *exprValue* be the result of performing Identifier Resolution as specified in 10.3.1 using *IdentifierName*.
3. Let *propValue* be GetValue(*exprValue*).
4. ReturnIfAbrupt(*propValue*).
5. Let *desc* be the Property Descriptor{[[Value]]: *propValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
6. Return the result of DefinePropertyOrThrow(*object*, *propName*, *desc*).

PropertyDefinition **:** *PropertyName* **:** *AssignmentExpression*

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *exprValue* be the result of evaluating *AssignmentExpression*.
4. Let *propValue* be GetValue(*exprValue*).
5. ReturnIfAbrupt(*propValue*).
6. Let *desc* be the Property Descriptor{[[Value]]: *propValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
7. Return the result of DefinePropertyOrThrow(*object*, *propName*, *desc*).

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

11.1.6 Function Defining Expressions

See 13.1 for *PrimaryExpression* **:** *FunctionExpression*.

See 13.4 for *PrimaryExpression* **:** *GeneratorExpression*.

See 13.5 for *PrimaryExpression* **:** *ClassExpression*.

11.1.7 Generator Comprehensions

Syntax

GeneratorComprehension **:**
  (*Comprehension* **)**

11.1.7.1 Static Semantics

Static Semantics: Early Errors

GeneratorComprehension **: (** *Comprehension* **)**

- It is a Syntax Error if *Comprehension* Contains *YieldExpression* is **true**.

11.1.7.2 Runtime Semantics

Runtime Semantics: Evaluation

GeneratorComprehension **: (** *Comprehension* **)**

Deleted: The production
Deleted: The production
Deleted: *PropertyNameAndValueList…ropertyDefinitionL*
Deleted: .
Deleted: <#>Let *obj* be the result of evaluating
Deleted: <#>Let *previous* be the result of calling the
Deleted: *success*
Deleted: P…rforming Property Definition Evaluation
Deleted: *PropertyNameAndValueList*
Deleted: Call the [[DefineOwnProperty]] internal metho
Deleted: If *successstatus* is an abrupt completion, return
Deleted: P…erforming Property Definition Evaluation
Deleted: *PropertyAssignment*
Deleted: Return *obj*.
Deleted: ¶
Deleted: The production
Deleted: *PropertyAssignment*
Deleted: is evaluated as follows:
Deleted: PropName(
Deleted:)
Deleted: as the *Identifier*
Deleted: If *propValue* is an abrupt completion, return
Deleted: Call
Deleted: calling the [[…efineOwn…roperty]]…rThrow(
Deleted: ,
Deleted: and
Deleted: , and **false**.
Deleted: <#>Return Property Identifier (*propName*, *des*
Deleted: The production
Deleted: *PropertyAssignment*
Deleted: is evaluated as follows:
Deleted: the result of evaluating
Deleted: PropName(of
Deleted:)
Deleted: If *propValue* is an abrupt completion, return
Deleted: calling Call the [[…efineOwn…roperty]] intern
Deleted: ,
Deleted: and
Deleted: , and **false**
Deleted: Return Property Identifier (*propName*, *desc*).
Comment [AWB665]: Need a actual reference
Deleted: <#>*PropertyDefinition* **:** *MethodDefinition*
Deleted: <#>*PropertyAssignment* **:** *PropertyName*
Deleted: <#>Runtime Semantics: Evaluation¶
Deleted: (*Expression* *ComprehensionForList*)
Deleted:)

1. If *GeneratorComprehension* is contained in strict mode code, then let *strict* be **true**; otherwise let *strict* be **false**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *parameters* be the production: *FormalParameters* **:** [empty].
4. Using *Comprehension* from the production that is being evaluated, let *body* be the supplemental syntactic grammar production: *GeneratorBody* **:** *Comprehension*.
5. Let *closure* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Arrow, *parameters*, *body*, *scope*, and *strict*.
6. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
7. Perform the abstract operation MakeConstructor with arguments *closure*, **true**, and *prototype*.
8. Let *iterator* be the result of calling the [[Call]] internal method of *closure* with **undefined** as *thisArgument* and a empty List as *argumentsList*.
9. Return *iterator*.

NOTE The GeneratorFunction object created in step 5 is not observable from ECMAScript code so an implementation may choose to avoid its allocation and initialization. In either case use other semantically equivalent means must be used to allocate and initialise the *iterator* object in step 8. In either case, the *prototype* object created in step 6 must be created because it is potentially observable as the value of the *iterator* object's [[Prototype]] internal data property.

11.1.8 Regular Expression Literals

Syntax

See 7.8.4.

Static Semantics

Static Semantics: Early Errors

PrimaryExpression **:** *RegularExpressionLiteral*

- It is a Syntax Error if BodyText of *RegularExpressionLiteral* cannot be recognised using the goal symbol *Pattern* of the ECMAScript RegExp grammar specified in 15.10.
- It is a Syntax Error if FlagText of *RegularExpressionLiteral* contains any character other than `"g"`, `"i"`, `"m"`, `"u"`, or `"y"`, or if it contains the same character more than once.

Runtime Semantics

Runtime Semantics: Evaluation

PrimaryExpression **:** *RegularExpressionLiteral*

1. A regular expression literal evaluates to a value of the Object type that is an instance of the standard built-in constructor RegExp. This value is determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpreted into two Strings Pattern and Flags, respectively. Then each time the literal is evaluated, a new object is created as if by the expression `new RegExp(Pattern, Flags)` where RegExp is the standard built-in constructor with that name. The newly constructed object becomes the value of the *RegularExpressionLiteral*.

11.1.9 Template Literals

Syntax

TemplateLiteral **:**
  NoSubstitutionTemplate
  TemplateHead *Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

Deleted: ize

Deleted: 5

Deleted:

Deleted: z

Comment [AWB866]: Should convert to a multistep algorithm and breakout a static semantic rule for the early error

Deleted: Quasi

Deleted: *Quasi*

Deleted: *FullQuasi*

Deleted: *Quasi*

Deleted: *QuasiMiddleList*$_{opt}$ [Lexical goal *InputElementQuasiTail*]

Deleted: *Quasi*

Deleted: *Tail*

TemplateSpans **:**
  TemplateTail
  TemplateMiddleList [Lexical goal *InputElementTemplateTail*] *TemplateTail*

TemplateMiddleList **:**
  TemplateMiddle Expression
  TemplateMiddleList [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

Deleted: *Quasi*

Deleted: [Lexical goal *InputElementQuasiTail*]

Static Semantics

Static Semantics: TemplateStrings

Deleted: QuasiStrings

With parameter *raw*.

TemplateLiteral **:** *NoSubstitutionTemplate*

Deleted: *Quasi…emplateLiteral* **:** *FullQuasi*

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*.

Deleted: QV…V of *FullQuasi*

Deleted:

Deleted: QRV…RV of *FullQuasi*

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

Deleted: *Quasi…emplateLiteral* **:** *Quasi…emplateHead*

1. If *raw* is **false**, then
   a. Let *head* be the TV of *TemplateHead*.
2. Else,
   a. Let *head* be the TRV of *TemplateHead*.
3. Let *tail* be TemplateStrings of *TemplateSpans* with argument *raw*.
4. Return a List containing *head* followed by the element, in order of *tail*.

Deleted: QV…V of *Quasi*

Deleted: <#>Let *tail* be the QV of *QuasiTail*.¶

Deleted: QRV…RV of *Quasi*

Deleted: the QRV …f *Quasi…emplateTail*

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* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

Deleted: *Quasi…emplateLiteral…pans* **:** *QuasiHead*

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*.

Deleted: <#>Let *head* be the QV of *QuasiHead*.¶

Deleted: <#>Let *head* be the QRV of *QuasiHead*.¶

Deleted: <#>Let *middle* be QuasiStrings of *QuasiMiddleList* with argument *raw*.¶

TemplateMiddleList **:** *TemplateMiddle Expression*

Deleted: *Quasi…emplateMiddleList* **:** [Lexical goal

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*.

Deleted: QV…V of *Quasi*

Deleted: QRV…RV of *Quasi*

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

Deleted: *Quasi…emplateMiddleList* **:**

1. Let *front* be TemplateStrings of *TemplateMiddleList* with argument *raw*.
2. If *raw* is **false**, then

Deleted: QuasiStrings…emplateStrings of *Quasi*

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*.

Runtime Semantics

Runtime Semantics: ArgumentListEvaluation

TemplateLiteral **:** *NoSubstitutionTemplate*

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

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

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

Runtime Semantics: GetTemplateCallSite Abstract Operation

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

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

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

Deleted: QV…V of *Quasi*
Deleted: QRV…RV of *Quasi*
Deleted: *Quasi…emplateLiteral* **:** *FullQuasi*
Deleted: the
Deleted: ion
Deleted: Quasi…emplateCallSite passing this
Deleted: *QuasiLiteral* **:** *QuasiHead Expression* [Lexical goal *InputElementQuasiTail*] *QuasiTail*
<#>Let *siteObj* be the the result of the abstracti operation GetQuasiCallSite passing this *QuasiLiteral* production as the argument.¶
<#>Let *firstSub* be the result of evaluating *Expression*. ¶
<#>ReturnIfAbrupt(*sub*).¶
<#>Return a List whose first element is *siteObj*, whose second elements *sub*. ¶
Quasi…emplateLiteral **:** *Quasi…emplateHead*
Deleted: the
Deleted: ion
Deleted: Quasi…emplateCallSite passing this
Deleted: *Quasi…emplateMiddleList*
Deleted: Quasi…emplateCallSite is called wit
Deleted: *quasi*
Deleted:
Deleted: r
Deleted: QuasiStrings…emplateStrings of *qua*
Deleted: QuasiStrings…emplateStrings of *qua*
Deleted: ion
Deleted: ion
Deleted: ,
Deleted: }
Deleted: , and **false**
Deleted: value
Deleted: ,
Deleted: }
Deleted: , and **false**
Deleted: Call…erform SetIntegrityLevel(the
Deleted: abstract operation with argument
Deleted: ,…and Property Descriptor {[[Value
Deleted: Call the [[FreezeObject]] internal method of
Deleted: abstract operation with argument
Deleted: *quasi*
Deleted: n

NOTE 2 Each *TemplateLiteral* in the program code is associated with a unique Template call site object that is used in the evaluation of tagged Templates (11.2.6). The same call site object is used each time a specific tagged Template is evaluated. Whether call site objects are created lazily upon first evaluation of the *TemplateLiteral* or eagerly prior to first evaluation is an implementation choice that is not observable to ECMAScript code.

Runtime Semantics: SubstitutionEvaluation

TemplateSpans **:** *TemplateTail*

1. Return an empty List.

TemplateSpans **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

1. Return the result of SubstitutionEvaluation of *TemplateMiddleList*.

TemplateMiddleList **:** *TemplateMiddle Expression*

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

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

1. Let *preceeding* be the result of SubstitutionEvaluation of *TemplateMiddleList* .
2. ReturnIfAbrupt(*preceeding*).
3. Let *next* be the result of evaluating *Expression*.
4. ReturnIfAbrupt(*next*).
5. Append *next* as the last element of the List *preceeding*.
6. Return *preceeding*.

Runtime Semantics: Evaluation

TemplateLiteral **:** *NoSubstitutionTemplate*

1. Return the string value whose elements are the TV of *NoSubstitutionTemplate* as defined in 7.8.5.

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

1. Let *head* be the TV of *TemplateHead* as defined in 7.8.5.
2. Let *sub* be the result of evaluating *Expression*.
3. Let *middle* be ToString(*sub*).
4. ReturnIfAbrupt(*middle*).
5. Let *tail* be the result of evaluating *TemplateSpans* .
6. ReturnIfAbrupt(*tail*).
7. Return the string value whose elements are the code units of *head* followed by the code units of *tail*.

TemplateSpans **:** *TemplateTail*

1. Let *tail* be the TV of *TemplateTail* as defined in 7.8.5.
2. Return the string whose elements are the code units of *tail*.

TemplateSpans **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

1. Let *head* be the result of evaluating *TemplateMiddleList*.
2. ReturnIfAbrupt(*head*).
3. Let *tail* be the TV of *TemplateTail* as defined in 7.8.5.
4. Return the string whose elements are the elements of *head* followed by the elements of *tail*.

TemplateMiddleList **:** *TemplateMiddle Expression*

Deleted: *Quasi*...*emplateLiteral* in the program code is

Deleted: *Quasi*...*emplateMiddleList* **:** [Lexical goal

Deleted: *middle*

Deleted: *Quasi*...*emplateMiddleList* **:**

Deleted: *Quasi*

Deleted: list

Deleted: *Quasi*...*emplateLiteral* **:** *FullQuasi*

Deleted: QV…V of *FullQuasi*

Deleted: 6

Deleted: *Quasi*...*emplateLiteral* **:** *Quasi*...*emplateHead*

Deleted: QV…V of *Quasi*

Deleted: 6

Comment [AWB967]: Note that the conversion semantics are like Stirng.prototype.concat rather than t + operator.

Deleted: <#>Let *tail* be the QV of *QuasiTail* as defined 7.8.6.¶

Deleted: 6

Deleted: *Quasi*...*emplateLiteral*...*pans* **:** *QuasiHead*

Comment [AWB968]: Note that the conversion semantics are like Stirng.prototype.concat rather than t + operator.

Deleted: <#>Let *head* be the QV of *QuasiHead* as defin in 7.8.6.¶
<#>Let *sub* be the result of evaluating *Expression*.¶
<#>Let *first* be ToString(*sub*).¶
<#>ReturnIfAbrupt(first).¶
<#>Let *rest* be the result of evaluating *QuasiMiddleList* .¶

Deleted: *rest*

Deleted: QV…V of *Quasi*

Deleted: 6

Deleted: code units

Deleted: followed by the elements of *first* followed by t code units of *rest*

Deleted: code units

Deleted: *Quasi*...*emplateMiddleList* **:** [Lexical goal

1. Let *head* be the TV of *TemplateMiddle* as defined in 7.8.5.
2. Let *sub* be the result of evaluating *Expression*.
3. Let *middle* be ToString(*sub*).
4. ReturnIfAbrupt(*middle*).
5. Return the sequence of characters consisting of the code units of *head* followed by the elements of *middle*.

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *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 7.8.5.
4. Let *sub* be the result of evaluating *Expression*.
5. Let *last* be ToString(*sub*).
6. ReturnIfAbrupt(*last*).
7. Return the sequence of characters consisting of the elements of *rest* followed by the code units of *middle* followed by the elements of *last*.

11.1.10 The Grouping Operator

Static Semantics: Early Errors

PrimaryExpression **:** *CoverParenthesisedExpressionAndArrowParameterList*

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

Static Semantics: IsValidSimpleAssignmentTarget

PrimaryExpression **:** *CoverParenthesisedExpressionAndArrowParameterList*

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

ParenthesisedExpression **:** **(** *Expression* **)**

1. Return IsValidSimpleAssignmentTarget of *Expression*.

Runtime Semantics: Evaluation

PrimaryExpression **:** *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *expr* be CoveredParenthesisedExpression of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return the result of evaluating *expr*.

ParenthesisedExpression **:** **(** *Expression* **)**

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

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

Deleted: QV
Deleted: *Quasi*
Deleted: 6
Comment [AWB969]: Note that the conversion semantics are like Stirng.prototype.concat rather than the + operator.
Deleted: *Quasi*
Deleted: *Quasi*
Deleted: *Quasi*
Deleted: *Quasi*
Deleted: *Quasi*
Deleted: QV
Deleted: *Quasi*
Deleted: 6
Comment [AWB970]: Note that the conversion semantics are like Stirng.prototype.concat rather than the + operator.
Deleted: <#>ComprehensionForList :¶ <#>ComprehensionFor ComprehensionForList ComprehensionFor¶ <#>ComprehensionFor :¶ <#>*for* (LeftHandSideExpression o… Expression)¶
Deleted: <#>The production *PropertyName* : *IdentifierName* is evaluat… as follows:¶ <#>Return the String value containing the same sequence of characters as the *IdentifierName*).¶ <#>The production *PropertyName* : *StringLiteral* is evaluated as follows:¶ <#>Return the SV of the *StringLiteral*.¶
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: *z*
Deleted: The production
Deleted: *z*
Deleted: *PrimaryExpression*
Deleted: is evaluated as follows:

11.2 Left-Hand-Side Expressions

Syntax

MemberExpression **:**
 [Lexical goal *InputElementRegExp*] *PrimaryExpression*
 MemberExpression **[** *Expression* **]**
 MemberExpression **.** *IdentifierName*
 MemberExpression *TemplateLiteral*
 super **[** *Expression* **]**
 super **.** *IdentifierName*
 new super *Arguments$_{opt}$*
 new *MemberExpression* *Arguments*

Deleted: *FunctionExpression*

Deleted: *Quasi*

Deleted: *MemberExpression* **<|** *TriangleLiteral*

Deleted: *MemberExpression* **<|** *SealedArrayLiteral*
MemberExpression **<|** *SealedObjectLiteral*
MemberExpression **<|** *FunctionExpression*
MemberExpression **<|** *ValueLiteral*

NewExpression **:**
 MemberExpression
 new *NewExpression*

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

Deleted: *Quasi*

Deleted: *CallExpression* **<|** *TriangleLiteral*

Arguments **:**
 ( )
 (*ArgumentList* **)**

ArgumentList **:**
 AssignmentExpression
 ... *AssignmentExpression*
 ArgumentList **,** *AssignmentExpression*
 ArgumentList **,** **...** *AssignmentExpression*

LeftHandSideExpression **:**
 NewExpression
 CallExpression

Deleted: *TriangleLiteral* **:**¶
SealedArrayLiteral
SealedObjectLiteral
FunctionExpression
ArrowFunction
ValueLiteral¶

Static Semantics

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**.

Deleted: c

Deleted:

Deleted:

MemberExpression **:** **super** **.** *IdentifierName*

1. If *symbol* is the *ReservedWord* **super**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.

3. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
4. Return **false**.

CallExpression **:** *CallExpression* **.** *IdentifierName*

1. If *CallExpression* Contains *symbol* is **true**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
4. Return **false**.

MemberExpression **:** **`new super`**

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

MemberExpression **:** **`new super`** *Arguments*

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

Static Semantics: IsValidSimpleAssignmentTarget

CallExpression **:**
  CallExpression **`[`** *Expression* **`]`**
  CallExpression **`.`** *IdentifierName*

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

1. Return **true**.

CallExpression **:**
  MemberExpression *Arguments*
  `super` *Arguments*
  CallExpression *Arguments*
  CallExpression *TemplateLiteral*

NewExpression **:** **`new`** *NewExpression*

MemberExpression **:**
  `new` **`super`** *Arguments*$_{opt}$
  `new` *MemberExpression* *Arguments*

1. Return **false**.

11.2.1 Property Accessors

Properties are accessed by name, using either the dot notation:

Deleted:

Deleted:

Deleted: c

Deleted:

Deleted:

Deleted: Static Semantics: IsInvalidAssignmentPattern¶
LeftHandSicdeExpression : *NewExpression*¶
<#>If the source code corresponding to *NewExpression* cannot be parsed as *ArrayLitera* then¶
<#>If the source code corresponding to *NewExpression* cannot be parsed as *ObjectLiteral*, then return **false**.¶
<#>If the source code corresponding to *NewExpression* cannot be parsed using *AssignmentPattern* (11.13.1) as the goal symbol return **true**.¶
<#>Return **false**.¶
LeftHandSicdeExpression : *CallExpression*¶
<#>Return **false**.¶

Deleted: *MemberExpression* *Arguments*
`super` *Arguments*
CallExpression *Arguments*

Comment [AWB1071]: These are false, because we disallow host functions returning reference values.

Deleted: *CallExpression* **:** *CallExpression* *TemplateLiteral* ¶

Deleted: *MemberExpression* **:** **`new`** *MemberExpression* *Arguments*¶

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 a string literal containing the same sequence of characters after processing of Unicode escape sequences as the *IdentifierName*.

Runtime Semantics: Evaluation

MemberExpression **:** *MemberExpression* **[** *Expression* **]**

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

CallExpression **:** *CallExpression* **[** *Expression* **]**

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

11.2.2 The `new` Operator

Runtime Semantics: Evaluation

NewExpression **:** **new** *NewExpression*

1. Let *ref* be the result of evaluating *NewExpression*.
2. Let *constructor* be GetValue(*ref*).
3. ReturnIfAbrupt(*constructor*).
4. If Type(*constructor*) is not Object, throw a **TypeError** exception.
5. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
6. Return the result of calling the [[Construct]] internal method on *constructor* with an empty List as the argument.

Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *baseValue* is an abrupt completion, return *baseValue*
Deleted: If *propertyNameValue* is an abrupt completion, return *propertyNameValue*
Deleted: Call
Deleted: If the result of
Deleted:
Deleted: *CheckObjectCoercible(baseValue)*
Deleted: is an abrupt completion, return that result
Deleted: contained in
Deleted: *baseValue*
Deleted: mode
Deleted: The production
Deleted: is
Deleted: ,
Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *constructor* is an abrupt completion, return *constructor*
Deleted: , providing no arguments (that is,
Deleted: list
Deleted: of
Comment [AWB972]: TODO probably need to do something about new operators in tail position.
Deleted: arguments)

MemberExpression : **new** *MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Let *constructor* be GetValue(*ref*).
3. ReturnIfAbrupt(*constructor*).
4. Let *argList* be the result of evaluating *Arguments*, producing an internal List of argument values (11.2.4).
5. ReturnIfAbrupt(*argList*).
6. If Type(*constructor*) is not Object, throw a **TypeError** exception.
7. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
8. Return the result of calling the [[Construct]] internal method on *constructor*, passing *argList* as the argument.

11.2.3 Function Calls

Runtime Semantics: Evaluation

CallExpression : *MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. If this *CallExpression* is in a tail position (13.6) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *ref*, *Arguments*, and *tailCall*.

CallExpression : *CallExpression Arguments*

1. Let *ref* be the result of evaluating *CallExpression*.
2. If this *CallExpression* is in a tail position (13.6) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *ref*, *Arguments*, and *tailCall*.

Runtime Semantics: EvaluateCall Abstract Operation

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

1. If Type(*ref*) is Reference, then
   a. If IsPropertyReference(*ref*) is **true**, then
      i. Return the result of the abstract operation EvaluateMethodCall with arguments *ref*, *arguments*, and *tailPosition*.
   b. Else, the base of *ref* is an Environment Record
      i. Let *thisValue* be the result of calling the WithBaseObject concrete method of GetBase(*ref*).
      ii. If *thisValue* is not **undefined**, then
         1. Let *newRef* be a value of type Reference whose base value is *thisValue* and whose referenced name is GetReferencedName(*ref*), and whose strict reference flag is IsStrictReference(ref).
         2. Return the result of the abstract operation EvaluateMethodCall with arguments *newRef*, *arguments*, and *tailPosition*.
2. Else Type(*ref*) is not Reference,
   a. Let *thisValue* be **undefined**.
3. Assert: This is a direct function call rather than a method call.
4. Let *func* be GetValue(*ref*).
5. ReturnIfAbrupt(*func*).
6. Let *argList* be the result of performing ArgumentListEvaluation of *arguments*.
7. ReturnIfAbrupt(*argList*).
8. If Type(*func*) is not Object, throw a **TypeError** exception.
9. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
10. If *tailPosition* is **true**, then perform the PrepareForTailCall abstract operation.

Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *constructor* is an abrupt completio return *constructor*
Deleted: list
Deleted: If *argList* is an abrupt completion, return *argList*
Deleted: providing the list
Deleted: values
Comment [AWB773]: TODO: tail calls.

Jan 19 meeting notes: Tentative decision is t support tail calls in strict mode only.

Deleted: The production
Deleted: is evaluated as follows:
Deleted: 7
Deleted:
Deleted: and
Deleted: 7
Deleted:
Deleted: and
Moved down [1]: A tail position call must either release any transient internal resource associated with the currently executing functi execution context before invoking the target function or reuse those resources in support the target function. ¶
NOTE 1 For example, a tail position call sho only grow an implementation's activation record stack by the amount that the size of th target function's activation record exceeds th size of the calling function's activation record the target function's activation record is small then the total size of the stack should decrea
Deleted:
Deleted: Reference,
Comment [AWB1574]: Explicit property references are handled as method calls.
Comment [AWB1575]: Implicit property references (via a with binding or global objec binding) are also handled as method calls.
Deleted: <#>Let *ref* be the result of evaluatin *MemberExpression*.¶
Deleted: If *func* is an abrupt completion, retur *func*
Deleted: evaluating
Deleted: *Arguments*
Deleted: , producing an internal list of argume values (see 11.2.4)
Deleted: If *argList* is an abrupt completion, return *argList*
Deleted: <#>If Type(*ref*) is Reference, then¶
<#>If IsPropertyReference(*ref*) is **true**, then
<#>Let *thisValue* be GetBaseGetThisValue(*ref*).¶
Deleted: then

11. Let *result* be the result of calling the [[Call]] internal method on *func*, passing *thisValue* as the *thisArgument* and *argList* as the *argumentsList*.
12. Assert: If *tailPosition* is **true**, the above call will not return here, but instead evaluation will continue with the resumption of *leafCallerContext* as the running execution context.
13. Assert: If result is not an abrupt completion then Type(*result*) is an ECMAScript language type
14. Return *result*.

Runtime Semantics: EvaluateMethodCall Abstract Operation

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

1. Assert: Type(*ref*) is Reference and IsPropertyReference(*ref*) is **true**
2. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
3. If HasPrimitiveBase(*ref*) is **true**, then
    a. Assert: In this case, *base* will never be **null** or **undefined**.
    b. Let *base* be ToObject(*base*).
4. Let *argList* be the result of performing ArgumentListEvaluation of *arguments*.
5. ReturnIfAbrupt(*argList*).
6. Let *base* be the result of calling GetBase(*ref*).
7. Let *thisValue* be GetThisValue(*ref*).
8. Let *key* be GetReferencedKey(*ref*).
9. If *tailPosition* is **true**, then perform the PrepareForTailCall abstract operation.
10. Let *result* be the result of calling the [[Invoke]] internal method on *base*, passing *key*, *argumentsList*, and *thisValue*.
11. Assert: If *tailPosition* is **true**, the above [[Invoke]] will not return here, but instead evaluation will continue with the resumption of *leafCallerContext* as the running execution context.
12. Assert: If result is not an abrupt completion then Type(*result*) is an ECMAScript language type
13. Return *result*.

11.2.4 The `super` Keyword

Static Semantics

Static Semantics: Early Errors

MemberExpression **:**
  `super` **`[`** *Expression* **`]`**
  `super` **`.`** *IdentifierName*
  `new` **`super`** *Arguments*$_{opt}$

CallExpression **:** **`super`** *Arguments*

- It is a Syntax Error if the source code parsed with this production is global code that is not eval code.
- It is a Syntax Error if the source code parsed with this production is eval code and the source code is not being processed by a direct call to eval that is contained in function code.

Runtime Semantics: Evaluation

MemberExpression **:** **`super`** **`[`** *Expression* **`]`**

1. Let *propertyNameReference* be the result of evaluating *Expression*.
2. Let *propertyNameValue* be GetValue(*propertyNameReference*).
3. Let *propertyKey* be ToPropertyKey(*propertyNameValue*).
4. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
5. Return the result of MakeSuperReference(*propertyKey*, *strict*).

MemberExpression **:** **`super`** **`.`** *IdentifierName*

Deleted: <#>Let *leafContext* be the running executio… context.¶
<#>Let *leafCallerContext* be *leafContext's* PreviousContext¶
<#>Suspend *leafContext*.¶
<#>Pop *leafContext* from the execution context contex… stack. Make *leafCallerContext* tThe execution context n… on the top of the stack becomes the running execution context, however it remains in its suspended state.¶
<#>Assert: *leafContext* has no further use. It will neve… be activated as the running execution context.¶

Deleted: Return

Deleted: providing

Deleted: as the **this** value

Deleted: providing the list

Deleted: *argument values*

Deleted: .

Comment [AWB1576]: This takes cares of property access on primitive values.

Comment [AWB1577]: It translate them into calls o… the [[MethodCall]] internal method of the the base obje…

Deleted: A tail position call must either release any transiaent internal resources associated with the curre… executing function execution context before invoking th… target function or reuse those resources in support of t… target function. ¶
NOTE 1 For example, a tail position call should only grow an implementation's activication record stack by t… amount that the size of the target function's activation record exceeds the size of the calling function's activat… record. If the target function's activation record is small… then the total size of the stack should decrease.¶

Deleted: The This production *CallExpression* **:** *CallExpression Arguments* is evaluated in exactly the sa… manner, as *CallExpression* **:** *MemberExpression Argumen…* except that the contained *CallExpression* is evaluated i… step 1.¶

Deleted: NOTE 2 . The returned result will never be o… type Reference if *func* is an native ordinary ECMAScrip… object. Whether calling an host exotic object can retur… value of type Reference is implementation-dependent.

Deleted: <#>It is a Syntax Error if the source code parsed with this production is global code code that

Deleted: <#>It is a Syntax Error if the source code parsed with this production is not extended code.¶

Deleted: code

Deleted: that is also global code

Deleted: <#>It is a Syntax Error if the source code parsed with this production is not extended code.¶

Deleted: <#>Let *env* be the result of performing the GetThisEnvironment abstract operation.¶

Deleted: <#>Let *baseReference* be the result of evaluati… *MemberExpression*.¶

Deleted: <#>ReturnIfAbrupt(CheckObjectCoercible(*ba… alue*)).Call CheckObjectCoercible(*baseValue*).¶

Deleted: *NameString*

Deleted: String

Deleted: a value of type Reference that is a Super Refere… whose base value is *baseValue* and, whose referenced nam…

1. Let *propertyKey* be StringValue of *IdentifierName*.
2. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
3. Return the result of MakeSuperReference(*propertyKey*, *strict*).

MemberExpression **:** `new super` *Arguments*$_{opt}$

1. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
2. Let *ref* be the result of MakeSuperReference(**undefined**, *strict*).
3. Let *constructor* be GetValue(*ref*).
4. ReturnIfAbrupt(*constructor*).
5. If *Arguments* is present, then
   a. Let *argList* be the result of evaluating *Arguments*, producing an internal List of argument values (11.2.4).
   b. ReturnIfAbrupt(*argList*).
6. Else,
   a. Let *argList* be a new empty List.
7. If Type(*constructor*) is not Object, throw a **TypeError** exception.
8. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
9. Return the result of calling the [[Construct]] internal method on *constructor*, passing *argList* as the argument.

CallExpression **:** `super` *Arguments*

1. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
2. Let *ref* be the result of MakeSuperReference(**undefined**, *strict*).
3. ReturnIfAbrupt(*ref*).
4. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
5. Return the result of the abstract operation EvaluateMethodCall with arguments *ref*, *Arguments*, and *tailCall*.

Runtime Semantics: Abstract Operation MakeSuperReference(propertyKey, strict)

1. Let *env* be the result of performing the GetThisEnvironment abstract operation.
2. If the result of calling the HasSuperBinding concrete method of *env* is **false**, then throw a **ReferenceError** exception.
3. Let *actualThis* be the result of calling the GetThisBinding concrete method of *env*.
4. Let *baseValue* be the result of calling the GetSuperBase concrete method of *env*.
5. Let *bv* be CheckObjectCoercible(*baseValue*).
6. ReturnIfAbrupt(*bv*).
7. If *propertyKey* is **undefined**, then
   a. Let *propertyKey* be the result of calling the GetMethodName concrete method of *env*.
8. Return a value of type Reference that is a Super Reference whose base value is *bv*, whose referenced name is *propertyKey*, whose thisValue is *actualThis*, and whose strict reference flag is *strict*.

11.2.5 Argument Lists

The evaluation of an argument list produces a List of values (see 8.7).

11.2.5.1 Runtime Semantics

Runtime Semantics: ArgumentListEvaluation

Arguments **:** `( )`

1. Return an empty List.

ArgumentList **:** *AssignmentExpression*

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be GetValue(*ref*).
3. ReturnIfAbrupt(*arg*).
4. Return a List whose sole item is *arg*.

ArgumentList **:** **. . .** *AssignmentExpression*

1. Let *list* be an empty List.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadValue* be GetValue(*spreadRef*).
4. Let *spreadObj* be ToObject(*spreadValue*).
5. ReturnIfAbrupt(*spreadObj*).
6. Let *lenVal* be the result of calling Get(*spreadObj*, `"length"`).
7. Let *spreadLen* be ToUint32(*lenVal*).
8. ReturnIfAbrupt(*spreadLen*).
9. Let *n* = 0.
10. Repeat, while *n* < *spreadLen*
    a. Let *nextArg* be the result of calling Get(*spreadObj*, ToString(*n*)).
    b. ReturnIfAbrupt(*nextArg*).
    c. Append *nextArg* as the last element of *list*.
    d. Let *n* = *n*+1.
11. Return *list*.

ArgumentList **:** *ArgumentList* **,** *AssignmentExpression*

1. Let *precedingArgs* be the result of evaluating *ArgumentList*.
2. ReturnIfAbrupt(*precedingArgs*).
3. Let *ref* be the result of evaluating *AssignmentExpression*.
4. Let *arg* be GetValue(*ref*).
5. ReturnIfAbrupt(*arg*).
6. Return a List whose length is one greater than the length of *precedingArgs* and whose items are the items of *precedingArgs*, in order, followed at the end by *arg* which is the last item of the new list.

ArgumentList **:** *ArgumentList* **,** **. . .** *AssignmentExpression*

1. Let *precedingArgs* be an empty List.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadValue* be GetValue(*spreadRef*).
4. Let *spreadObj* be ToObject(*spreadValue*).
5. ReturnIfAbrupt(*spreadObj*).
6. Let *lenVal* be the result of calling Get(*spreadObj*, `"length"`).
7. Let *spreadLen* be ToUint32(*lenVal*).
8. ReturnIfAbrupt(*spreadLen*).
9. Let *n* = 0.
10. Repeat, while *n* < *spreadLen*
    a. Let *nextArg* be the result of calling Get(*spreadObj*, ToString(*n*)).
    b. ReturnIfAbrupt(*nextArg*).
    c. Append *nextArg* as the last element of *precedingArgs*.
    d. Let *n* = *n*+1.
11. Return *precedingArgs*.

Deleted: The production *Arguments* **:** **(** *ArgumentList* **)** evaluated as follows:¶
<#>Return the result of evaluating *ArgumentList*.¶

Deleted: The production

Deleted: is evaluated as follows:

Deleted: If *arg* is an abrupt completion, return *arg*

Deleted: ...

Comment [AW78]: Note that the value the spread operator is applied to is coerced to an Object.

Deleted: If *spreadObj* is an abrupt completion, return *spreadObj*

Deleted: the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: "

Deleted: "

Deleted: the [[

Deleted:]] internal method of

Deleted: passing

Deleted:

Deleted: as the argument

Deleted: The production

Deleted: is evaluated as follows:

Deleted: If *precedingArgs* is an abrupt completion, return *precedingArgs*

Deleted: If *arg* is an abrupt completion, return *arg*

Deleted: The production

Deleted: ...

Deleted: is evaluated as follows:

Comment [AW79]: Note that the value the spread operator is applied to is coerced to an Object.

Deleted: If *spreadObj* is an abrupt completion, return *spreadObj*

Deleted: the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: "

Deleted: "

Deleted: the [[

Deleted:]] internal method of

Deleted:

Deleted: passing

Deleted: as the argument

11.2.6 Tagged Templates

11.2.6.1 Runtime Semantics

Runtime Semantics: Evaluation

MemberExpression : *MemberExpression TemplateLiteral*

1. Let *tagRef* be the result of evaluating *MemberExpression*.
2. If this *MemberExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *tagRef*, *TemplateLiteral*, and *tailCall*.

CallExpression **:** *CallExpression TemplateLiteral*

1. Let *tagRef* be the result of evaluating *CallExpression*.
2. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *tagRef*, *TemplateLiteral*, and *tailCall*.

11.3 Postfix Expressions

Syntax

PostfixExpression **:**
 LeftHandSideExpression
 LeftHandSideExpression [no *LineTerminator* here] `++`
 LeftHandSideExpression [no *LineTerminator* here] `--`

Static Semantics

Static Semantics: Early Errors

PostfixExpression **:**
 LeftHandSideExpression [no *LineTerminator* here] `++`
 LeftHandSideExpression [no *LineTerminator* here] `--`

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

Static Semantics: IsValidSimpleAssignmentTarget

PostfixExpression **:**
 LeftHandSideExpression [no *LineTerminator* here] `++`
 LeftHandSideExpression [no *LineTerminator* here] `--`

1. Return **false**.

11.3.1 Postfix Increment Operator

Runtime Semantics: Evaluation

PostfixExpression **:** *LeftHandSideExpression* [no *LineTerminator* here] `++`

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be ToNumber(GetValue(*lhs*)).
3. ReturnIfAbrupt(*oldValue*).
4. Let *newValue* be the result of adding the value `1` to *oldValue*, using the same rules as for the `+` operator (see 11.6.3).
5. Let *status* be PutValue(*lhs*, *newValue*).

Deleted: Quasi

Deleted: The runtime semantics of productio
CallExpression **:** *CallExpression QuasiTemplateLiteral*¶
is identical to that of¶
MemberExpression **:** *MemberExpression QuasiTemplateLiteral*¶
but with evaluation of *CallExpression* substitut
for the evaluation of *MemberExpression*.¶

Deleted: *Quasi*

Deleted: *Quasi*

Deleted: *PostfixExpression* **:**¶
LeftHandSideExpression
LeftHandSideExpression [no *LineTerminator* here] `++`
LeftHandSideExpression [no *LineTerminator* here] `--`¶
<#>It is a Syntax Error if the derived *LeftHandSideExpression* is *PrimaryExpressio* *ObjectLiteral* and ContainsNonObjectLiteralProductions of *ObjectLiteral* is **true**.¶
<#>It is a Syntax Error if the derived *LeftHandSideExpression* is *PrimaryExpressio* `(` *Expression* `)` and *Expression* derived a production that if used in place of *LeftHandSideExpression* would produce a Syntax Error according to these rules. This rule is recursively applied.¶

Deleted: It is a Syntax Error if the *PostfixExpression* is contained in strict code and *LeftHandSideExpression* is the *Identifier* `eval` or the *Identifier* `arguments`.

Deleted: ¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in extend code and the *LeftHandSideExpression* is a *Literal* or a *FunctionExpression*.¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in extend code and the *LeftHandSideExpression* is an *Identifier* that does not statically resolve to declarative environment record binding or the resolved binding is an immutable bindi
It is a Syntax Error if the *LeftHandSideExpression* is *PrimaryExpressio* `(` *Expression* `)` and *Expression* derived a production that if used in place of *LeftHandSideExpression* that would produce Syntax Error according to these rules. This rule is recursively applied.

Deleted: The production

Deleted: is evaluated as follows:

Deleted: <#>Throw a **SyntaxError** exceptio the following conditions are all true:¶
<#>Type(*lhs*) is Reference is **true**¶
<#>IsStrictReference(*lhs*) is **true**¶
<#>Type(GetBase(*lhs*)) is Environment Reco
<#>GetReferencedName(*lhs*) is either `"eval` or `"arguments"`¶

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

Deleted: Call

Deleted: If

6. ReturnIfAbrupt(*status*).
7. Return *oldValue*.

11.3.2 Postfix Decrement Operator

Runtime Semantics: Evaluation

PostfixExpression **:** *LeftHandSideExpression* [no *LineTerminator* here] `--`

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. Let *oldValue* be ToNumber(GetValue(*lhs*)).
3. Let *newValue* be the result of subtracting the value `1` from *oldValue*, using the same rules as for the `-` operator (11.6.3).
4. Let *status* be PutValue(*lhs*, *newValue*).
5. ReturnIfAbrupt(*status*).
6. Return *oldValue*.

Deleted: The production

Deleted: is evaluated as follows:

Deleted: <#>Throw a **SyntaxError** exception if the following conditions are all true:¶
<#>Type(*lhs*) is Reference is **true**¶
<#>IsStrictReference(*lhs*) is **true**¶
<#>Type(GetBase(*lhs*)) is Environment Record¶
<#>GetReferencedName(*lhs*) is either `"eval"` or `"arguments"`¶

Deleted: If Call

11.4 Unary Operators

Syntax

UnaryExpression **:**
  PostfixExpression
  `delete` *UnaryExpression*
  `void` *UnaryExpression*
  `typeof` *UnaryExpression*
  `++` *UnaryExpression*
  `--` *UnaryExpression*
  `+` *UnaryExpression*
  `-` *UnaryExpression*
  `~` *UnaryExpression*
  `!` *UnaryExpression*

Static Semantics

Static Semantics: Early Errors

UnaryExpression **:**
  `++` *UnaryExpression*
  `--` *UnaryExpression*

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

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

Deleted: *UnaryExpression* **:**¶
`delete` *UnaryExpression*¶
<#>
++ UnaryExpression
-- UnaryExpression¶
<#>It is a Syntax Error if the *UnaryExpression* is contained in strict code and the derived *UnaryExpression* is the Identifier `eval` or the Identifie `arguments`.¶
<#>It is a Syntax Error if the *UnaryExpression* is contained in extended code and the derived *UnaryExpression* is a *Literal* or a *FunctionExpression*.¶
<#>It is a Syntax Error if the *UnaryExpression* is contained in extended code and the derived *UnaryExpression* is an *Identifier* that does not staticall resolve to a declarative environment record binding if the resolved binding is an immutable binding.¶
<#>It is a Syntax Error if the derived *UnaryExpressio* is
PrimaryExpression : (*CoverParenthesizedExpressionAndArrowParameterList* *Expression*)
and *Expression* derives a production that if used in place of *LeftHandSideUnaryExpression* would produce Syntax Error according to these rules. This rule is recursively applied.¶

11.4.1 The `delete` Operator

Static Semantics: Early Errors

UnaryExpression **:** `delete` *UnaryExpression*

- It is a Syntax Error if the *UnaryExpression* is contained in strict code and the derived *UnaryExpression* is *PrimaryExpression : Identifier*.
- It is a Syntax Error if the derived *UnaryExpression* is
  PrimaryExpression : CoverParenthesisedExpressionAndArrowParameterList
  and derives a production that, if used in place of *UnaryExpression*, would produce a Syntax Error according to these rules. This rule is recursively applied.

NOTE The last rule means that expressions such as

```
delete (((foo)))
```

produce early errors because of recursive application of the first rule.

Runtime Semantics: Evaluation

UnaryExpression **:** `delete` *UnaryExpression*

1. Let *ref* be the result of evaluating *UnaryExpression*.
2. ReturnIfAbrupt(*ref*).
3. If Type(*ref*) is not Reference, return **true**.
4. If IsUnresolvableReference(*ref*) is **true**, then,
   a. If IsStrictReference(*ref*) is **true**, then throw a **SyntaxError** exception.
   b. Return **true**.
5. If IsPropertyReference(*ref*) is **true**, then
   a. If IsSuperReference(*ref*), then throw a **ReferenceError** exception.
   b. Let *deleteStatus* be the result of calling the [[Delete]] internal method on ToObject(GetBase(*ref*)), providing GetReferencedName(*ref*) as the argument.
   c. ReturnIfAbrupt(*deleteStatus*).
   d. If *deleteStatus* is **false** and IsStrictReference(*ref*) is **true**, then throw a **TypeError** exception.
   e. Return *deleteStatus*.
6. Else *ref* is a Reference to an Environment Record binding,
   a. Let *bindings* be GetBase(*ref*).
   b. Return the result of calling the DeleteBinding concrete method of *bindings*, providing GetReferencedName(*ref*) as the argument.

NOTE When a `delete` operator occurs within strict mode code, a **SyntaxError** exception is thrown if its *UnaryExpression* is a direct reference to a variable, function argument, or function name. In addition, if a `delete` operator occurs within strict mode code and the property to be deleted has the attribute { [[Configurable]]: **false** }, a **TypeError** exception is thrown.

11.4.2 The `void` Operator

Runtime Semantics: Evaluation

UnaryExpression **:** `void` *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *status* be GetValue(*expr*).
3. ReturnIfAbrupt(*status*).
4. Return **undefined**.

NOTE GetValue must be called even though its value is not used because it may have observable side-effects.

Deleted: derives an
Deleted: that statically resolves to a environment record
Deleted: *z*
Deleted: is
Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *ref* is an abrupt completion, return
Deleted: Else, r
Deleted: Return
Deleted: and IsStrictReference(*ref*)
Deleted: s
Deleted: t
Deleted: true
Deleted: ,
Deleted: so
Deleted: <#>If IsStrictReference(*ref*) is tr throw a **SyntaxError** exception.¶
Deleted: The production
Deleted: is evaluated as follows:
Deleted: Call
Deleted: <#>ReturnIfAbrupt(*exprref*).If *ref* i abrupt completion, return *ref*.¶

11.4.3 The `typeof` Operator

Runtime Semantics: Evaluation

UnaryExpression **:** **`typeof`** *UnaryExpression*

1. Let *val* be the result of evaluating *UnaryExpression*.
2. If Type(*val*) is Reference, then
    a. If IsUnresolvableReference(*val*) is **true**, return **`"undefined"`**.
    b. Let *val* be GetValue(*val*).
3. ReturnIfAbrupt(*val*).
4. Return a String determined by Type(*val*) according to **Table 31**.

Table 31 — typeof Operator Results

***Type of** val*	***Result***
Undefined	**`"undefined"`**
Null	**`"object"`**
Boolean	**`"boolean"`**
Number	**`"number"`**
String	**`"string"`**
Object (ordinary and does not implement [[Call]])	**`"object"`**
Object (is a Symbol exotic object)	**`"symbol"`**
Object (standard exotic other than Symbol and does not implement [[Call]])	**`"object"`**
Object (implements [[Call]])	**`"function"`**
Object (non-standard exotic and does not implement [[Call]])	Implementation-defined. May not be **`"undefined"`**, **`"boolean"`**, **`"number"`**, **`"symbol"`**, or **`"string"`**.

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

11.4.4 Prefix Increment Operator

Runtime Semantics: Evaluation

UnaryExpression **:** **`++`** *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. Let *newValue* be the result of adding the value **`1`** to *oldValue*, using the same rules as for the **`+`** operator (see 11.6.3).
5. Let *status* be PutValue(*expr*, *newValue*).
6. ReturnIfAbrupt(*status*).
7. Return *newValue*.

Deleted: The production

Deleted: is evaluated as follows:

Deleted: If *val* is an abrupt completion, return *val*.

Deleted: Table 31

Deleted: native

Deleted: native or host and does

Deleted:

Deleted: host

Deleted: unless explicitly specified

Deleted: except m

Deleted: The production

Deleted: is evaluated as follows:

Deleted: <#>Throw a **SyntaxError** exception if the following conditions are all true:¶
<#>Type(*expr*) is Reference is **true**¶
<#>IsStrictReference(*expr*) is **true**¶
<#>Type(GetBase(*expr*)) is Environment Record¶
<#>GetReferencedName(*expr*) is either **`"eval"`** or **`"arguments"`**¶

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

Deleted: If

Deleted: Call

11.4.5 Prefix Decrement Operator

Runtime Semantics: Evaluation

UnaryExpression **:** **--** *UnaryExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. Let *newValue* be the result of subtracting the value **1** from *oldValue*, using the same rules as for the **-** operator (see 11.6.3).
5. Let *status* be PutValue(*expr*, *newValue*).
6. ReturnIfAbrupt(*status*).
7. Return *newValue*.

Deleted: <#>Throw a **SyntaxError** exceptio the following conditions are all true:¶
<#>Type(*expr*) is Reference is **true**¶
<#>IsStrictReference(*expr*) is **true**¶
<#>Type(GetBase(*expr*)) is Environment Rec
<#>GetReferencedName(*expr*) is either **"eva** or **"arguments"**¶

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

Deleted: Call

Deleted: If

11.4.6 Unary + Operator

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

Runtime Semantics: Evaluation

UnaryExpression **:** **+** *UnaryExpression*

Deleted: The production

Deleted: is evaluated as follows:

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

11.4.7 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**.

Deleted: Note that negating

Runtime Semantics: Evaluation

UnaryExpression **:** **–** *UnaryExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. If *oldValue* is **NaN**, return **NaN**.
5. Return the result of negating *oldValue*; that is, compute a Number with the same magnitude but opposite sign.

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

11.4.8 Bitwise NOT Operator (~)

Runtime Semantics: Evaluation

UnaryExpression **:** **~** *UnaryExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToInt32(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. Return the result of applying bitwise complement to *oldValue*. The result is a signed 32-bit integer.

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

11.4.9 Logical NOT Operator (!)

Runtime Semantics: Evaluation

UnaryExpression **:** **!** *UnaryExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToBoolean(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. If *oldValue* is **true**, return **false**.
5. Return **true**.

11.5 Multiplicative Operators

Syntax

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

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

Runtime Semantics: Evaluation

The production *MultiplicativeExpression* **:** *MultiplicativeExpression* @ *UnaryExpression*, where @ stands for one of the operators in the above definitions, is evaluated as follows:

1. Let *left* be the result of evaluating *MultiplicativeExpression*.
2. Let *leftValue* be GetValue(*left*).
3. ReturnIfAbrupt(*leftValue*).
4. Let *right* be the result of evaluating *UnaryExpression*.
5. Let *rightValue* be GetValue(*right*).
6. Let *lnum* be ToNumber(*leftValue*).
7. ReturnIfAbrupt(*lnum*).
8. Let *rnum* be ToNumber(*rightValue*).
9. ReturnIfAbrupt(*rnum*).
10. Return the result of applying the specified operation (*, /, or %) to *lnum* and *rnum*. See the Notes below 11.5.1, 11.5.2, 11.5.3.

11.5.1 Applying the `*` Operator

The `*` operator performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in ECMAScript, because of finite precision.

The result of a floating-point multiplication is governed by the rules of IEEE 754 binary double-precision arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Multiplication of an infinity by a zero results in **NaN**.
- Multiplication of an infinity by an infinity results in an infinity. The sign is determined by the rule already stated above.
- Multiplication of an infinity by a finite nonzero value results in a signed infinity. The sign is determined by the rule already stated above.

Deleted: If *oldValue* is an abrupt completion, return *oldValue*.

Deleted: Semantics¶

Deleted: *lval*

Deleted: If *leftValue* is an abrupt completion, return *leftValue*.

Deleted: *leftNum*

Deleted: *n*

Deleted: If *leftNum* is an abrupt completion, return *leftNu*

Deleted: *rightNum*

Deleted: *n*

Deleted: If *rightNum* is an abrupt completion, return *rightNum*.

Deleted: *leftNum*

Deleted: *rightNum*

- In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the result is then a zero of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

11.5.2 Applying the `/` Operator

The `/` operator performs division, producing the quotient of its operands. The left operand is the dividend and the right operand is the divisor. ECMAScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification of IEEE 754 arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Division of an infinity by an infinity results in **NaN**.
- Division of an infinity by a zero results in an infinity. The sign is determined by the rule already stated above.
- Division of an infinity by a nonzero finite value results in a signed infinity. The sign is determined by the rule already stated above.
- Division of a finite value by an infinity results in zero. The sign is determined by the rule already stated above.
- Division of a zero by a zero results in **NaN**; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.
- Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

11.5.3 Applying the `%` Operator

The `%` operator yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

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

The result of a floating-point remainder operation as computed by the `%` operator is not the same as the "remainder" operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a truncating division, and so its behaviour is not analogous to that of the usual integer remainder operator. Instead the ECMAScript language defines `%` on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function fmod.

The result of an ECMAScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is **NaN**.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is nonzero and finite, the result is the same as the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical

relation r = n − (d × q) where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d. r is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode.

11.6 Additive Operators

Syntax

AdditiveExpression **:**
- *MultiplicativeExpression*
- *AdditiveExpression* **+** *MultiplicativeExpression*
- *AdditiveExpression* **−** *MultiplicativeExpression*

Static Semantics: IsValidSimpleAssignmentTarget

AdditiveExpression **:**
- *AdditiveExpression* **+** *MultiplicativeExpression*
- *AdditiveExpression* **−** *MultiplicativeExpression*

1. Return **false**.

11.6.1 The Addition operator (+)

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

Formatted: Note

Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
12. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: If *lprim* is an abrupt completion, return *lprim*.

Deleted: If *rprim* is an abrupt completion, return *rprim*.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (11.8.1), by using the logical-or operation instead of the logical-and operation.

Deleted: 5

Deleted: 6

Deleted: native

Deleted: Host

Deleted: 7

Deleted: 3

Deleted: comparison

Deleted: for the relational operators

Deleted: 5

11.6.2 The Subtraction Operator (−)

Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **−** *MultiplicativeExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).

3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToNumber(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToNumber(*rval*).
10. ReturnIfAbrupt(*rnum*).
11. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 11.6.3.

Deleted: .If *lval* is an abrupt completion, retur *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: If *lnum* is an abrupt completion, retu *lnum*.

Deleted: If *rnum* is an abrupt completion, retu *rnum*.

11.6.3 Applying the Additive Operators to Numbers

The **+** operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The **-** operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 binary double-precision arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sum of two infinities of opposite sign is **NaN**.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two negative zeroes is **–0**. The sum of two positive zeroes, or of two zeroes of opposite sign, is **+0**.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is **+0**.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

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 $a + (–b)$.

11.7 Bitwise Shift Operators

Syntax

ShiftExpression **:**
    AdditiveExpression
    ShiftExpression **<<** *AdditiveExpression*
    ShiftExpression **>>** *AdditiveExpression*
    ShiftExpression **>>>** *AdditiveExpression*

Static Semantics: IsValidSimpleAssignmentTarget

ShiftExpression **:**
    ShiftExpression **<<** *AdditiveExpression*
    ShiftExpression **>>** *AdditiveExpression*
    ShiftExpression **>>>** *AdditiveExpression*

1. Return **false**.

11.7.1 The Left Shift Operator (<<)

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

Formatted: Note

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **<<** *AdditiveExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetVal(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*rnum*).
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
12. Return the result of left shifting *lnum* by *shiftCount* bits. The result is a signed 32-bit integer.

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: *n*

Deleted: If *lnum* is an abrupt completion, return *lnum*.

Deleted: *n*

Deleted: If *rnum* is an abrupt completion, return *rnum*.

11.7.2 The Signed Right Shift Operator (>>)

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

Formatted: Note

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **>>** *AdditiveExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*rnum*).
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
12. Return the result of performing a sign-extending right shift of *lnum* by *shiftCount* bits. The most significant bit is propagated. The result is a signed 32-bit integer.

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: *n*

Deleted: If *lnum* is an abrupt completion, return *lnum*.

Deleted: *n*

Deleted: If *rnum* is an abrupt completion, return *rnum*.

11.7.3 The Unsigned Right Shift Operator (>>>)

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

Formatted: Note

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **>>>** *AdditiveExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.

Deleted: If *lval* is an abrupt completion, return *lval*.

5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToUint32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*rnum*).
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
12. Return the result of performing a zero-filling right shift of *lnum* by *shiftCount* bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

Deleted: If *rval* is an abrupt completion, return *rval*.
Deleted: *n*
Deleted: If *lnum* is an abrupt completion, return *lnum*.
Deleted: *n*
Deleted: If *rnum* is an abrupt completion, return *rnum*.

11.8 Relational Operators

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

Formatted: Note

Syntax

RelationalExpression **:**
  ShiftExpression
  RelationalExpression **<** *ShiftExpression*
  RelationalExpression **>** *ShiftExpression*
  RelationalExpression **<=** *ShiftExpression*
  RelationalExpression **>=** *ShiftExpression*
  RelationalExpression **`instanceof`** *ShiftExpression*
  RelationalExpression **`in`** *ShiftExpression*

RelationalExpressionNoIn **:**
  ShiftExpression
  RelationalExpressionNoIn **<** *ShiftExpression*
  RelationalExpressionNoIn **>** *ShiftExpression*
  RelationalExpressionNoIn **<=** *ShiftExpression*
  RelationalExpressionNoIn **>=** *ShiftExpression*
  RelationalExpressionNoIn **`instanceof`** *ShiftExpression*

The semantics of the *RelationalExpressionNoIn* productions are the same as the *RelationalExpression* productions except that the contained *RelationalExpressionNoIn* is used in place of the contained *RelationalExpression*.

Deleted: evaluated in
Deleted: manner
Deleted: evaluated instead

NOTE The "NoIn" variants are needed to avoid confusing the `in` operator in a relational expression with the `in` operator in a `for` statement.

Static Semantics: IsValidSimpleAssignmentTarget

RelationalExpression **:**
  RelationalExpression **<** *ShiftExpression*
  RelationalExpression **>** *ShiftExpression*
  RelationalExpression **<=** *ShiftExpression*
  RelationalExpression **>=** *ShiftExpression*
  RelationalExpression **`instanceof`** *ShiftExpression*
  RelationalExpression **`in`** *ShiftExpression*

1. Return **false**.

11.8.1 Runtime Semantics

Runtime Semantics: Abstract Relational Comparison

The comparison *x* < *y*, where *x* and *y* are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). In addition to *x* and *y* the algorithm takes a Boolean flag named *LeftFirst* as a parameter. The flag is used to control the order in which operations with potentially visible side-effects are performed upon *x* and *y*. It is necessary because ECMAScript specifies left to right evaluation of expressions. The default value of *LeftFirst* is **true** and indicates that the *x* parameter corresponds to an expression that occurs to the left of the *y* parameter's corresponding expression. If *LeftFirst* is **false**, the reverse is the case and operations must be performed upon *y* before *x*. Such a comparison is performed as follows:

1. ReturnIfAbrupt(*x*).
2. ReturnIfAbrupt(*y*).
3. If the *LeftFirst* flag is **true**, then
   a. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
   b. ReturnIfAbrupt(*px*).
   c. Let *py* be the result of calling ToPrimitive(*y*, hint Number).
   d. ReturnIfAbrupt(*py*).
4. Else the order of evaluation needs to be reversed to preserve left to right evaluation
   a. Let *py* be the result of calling ToPrimitive(*y*, hint Number).
   b. ReturnIfAbrupt(*py*).
   c. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
   d. ReturnIfAbrupt(*px*).
5. If both *px* and *py* are Strings, then
   a. If *py* is a prefix of *px*, return **false**. (A String value *p* is a prefix of String value *q* if *q* can be the result of concatenating *p* and some other String *r*. Note that any String is a prefix of itself, because *r* may be the empty String.)
   b. If *px* is a prefix of *py*, return **true**.
   c. Let *k* be the smallest nonnegative integer such that the character at position *k* within *px* is different from the character at position *k* within *py*. (There must be such a *k*, for neither String is a prefix of the other.)
   d. Let *m* be the integer that is the code unit value for the character at position *k* within *px*.
   e. Let *n* be the integer that is the code unit value for the character at position *k* within *py*.
   f. If *m* < *n*, return **true**. Otherwise, return **false**.
6. Else,
   a. Let *nx* be the result of calling ToNumber(*px*). Because *px* and *py* are primitive values evaluation order is not important.
   b. Let *ny* be the result of calling ToNumber(*py*).
   c. If *nx* is **NaN**, return **undefined**.
   d. If *ny* is **NaN**, return **undefined**.
   e. If *nx* and *ny* are the same Number value, return **false**.
   f. If *nx* is **+0** and *ny* is **−0**, return **false**.
   g. If *nx* is **−0** and *ny* is **+0**, return **false**.
   h. If *nx* is +∞, return **false**.
   i. If *ny* is +∞, return **true**.
   j. If *ny* is −∞, return **false**.
   k. If *nx* is −∞, return **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 **true**. Otherwise, return **false**.

NOTE 1 Step 5 differs from step 11 in the algorithm for the addition operator **+** (11.6.1) in using and instead of or.

NOTE 2 The comparison of Strings uses a simple lexicographic ordering on sequences of code unit values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore String values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalised form. Also, note that for strings containing supplementary characters, lexicographic ordering on sequences of UTF-16 code unit values differs from that on sequences of code point values.

Deleted: <#>Semantics¶
<#>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.¶
<#>The *RelationalExpressionNoIn* productions are evaluated in the same manner as the *RelationalExpression* productions except that the contained *RelationalExpressionNoIn* is evaluated instead of the contained *RelationalExpression*.¶

Deleted: The

Deleted: Algorithm

Deleted: If *x* is an abrupt completion, return *x*.

Deleted: If *y* is an abrupt completion, return *y*.

Deleted: If px is an abrupt completion, return px.

Deleted: If *py* is an abrupt completion, return *py*.

Deleted: If *py* is an abrupt completion, return *py*.

Deleted: If px is an abrupt completion, return px.

Deleted: If it is not the case that

Deleted: both Type(*px*) is String and Type(*py*) is String, then

Deleted: <#>Else, both *px* and *py* are Strings,¶
<#>If *py* is a prefix of *px*, return **false**. (A String value 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.)¶
<#>If *px* is a prefix of *py*, return **true**.¶
<#>Let *k* be the smallest nonnegative integer such that character at position *k* within *px* is different from the character at position *k* within *py*. (There must be such a for neither String is a prefix of the other.)¶
<#>Let *m* be the integer that is the code unit value for t character at position *k* within *px*.¶
<#>Let *n* be the integer that is the code unit value for t character at position *k* within *py*.¶
<#>If *m* < *n*, return **true**. Otherwise, return **false**.¶

Deleted: 3

Deleted: 7

Runtime Semantics: Evaluation

RelationalExpression **:** *RelationalExpression* **<** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. Let *r* be the result of performing Abstract Relational Comparison *lval* < *rval*. (see 11.8.5)
7. ReturnIfAbrupt(*r*).
8. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression **:** *RelationalExpression* **>** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. Let *r* be the result of performing Abstract Relational Comparison *rval* < *lval* with *LeftFirst* equal to **false**.
7. ReturnIfAbrupt(*r*).
8. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression **:** *RelationalExpression* **<=** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. Let *r* be the result of performing Abstract Relational Comparison *rval* < *lval* with *LeftFirst* equal to **false**.
7. ReturnIfAbrupt(*r*).
8. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

RelationalExpression **:** *RelationalExpression* **>=** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. Let *r* be the result of performing Abstract Relational Comparison *lval* < *rval*.
7. ReturnIfAbrupt(*r*).
8. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

RelationalExpression **:** *RelationalExpression* `instanceof` *ShiftExpression*

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

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:

Deleted: <#>11.8.1 The Less-than Operator (<)¶
The production
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return *lval*.
Deleted: abstract
Deleted: relational
Deleted: comparison
Deleted: <#>*11.8.2 The Greater-than Operator (>)*¶
The production
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return
Deleted: abstract
Deleted: relational
Deleted: comparison
Deleted: . (see 11.8.5).
Deleted: <#>*11.8.3 The Less-than-or-equ*
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return
Deleted: abstract
Deleted: relational
Deleted: comparison
Deleted: . (see 11.8.5).
Deleted: <#>11.8.4 The Greater-than-
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return
Deleted: abstract
Deleted: relational
Deleted: comparison
Deleted: (see 11.8.5)
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return
Deleted: If *rval* is an abrupt completion, return
Deleted: <#>If Type(*rval*) is not Object, thro
Deleted: calling the
Deleted: calling the [[HasInstance
Deleted: O
Deleted: *abstract ooperator with arguments*
Deleted: *]] internal method of*
Deleted: *r*
Deleted: with argument
Deleted: and
Deleted: *lval*
Deleted: HasI
Deleted: ,

1. If Type(*C*) is not Object, throw a **TypeError** exception.
2. Let *instOfHandler* be the result of GetMethod(*C*,@@hasInstance).
3. ReturnIfAbrupt(*instOfHandler*).
4. If *instOfHandler* is not **undefined**, then
    a. Let *result* be the result of calling the [[Call]] internal method of *instOfHandler* passing *C* as *thisArgument* and a new List containing *O* as *argumentsList*.
    b. Return ToBoolean(*result*).
5. If IsCallable(*C*) is **false**, then throw a **TypeError** exception.
6. Return the result of OrdinaryHasInstance(*C*, *O*).

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

RelationalExpression **:** *RelationalExpression* **in** *ShiftExpression*

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

11.9 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 **:**
  RelationalExpression
  EqualityExpression **==** *RelationalExpression*
  EqualityExpression **!=** *RelationalExpression*
  EqualityExpression **===** *RelationalExpression*
  EqualityExpression **!==** *RelationalExpression*

EqualityExpressionNoIn **:**
  RelationalExpressionNoIn
  EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
  EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
  EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
  EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

The semantics of the *EqualityExpressionNoIn* productions are the same as the *EqualityExpression* productions except that the contained *EqualityExpressionNoIn* and *RelationalExpressionNoIn* are used in place of the contained *EqualityExpression* and *RelationalExpression*, respectively.

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

Deleted: Return

Deleted: it is

Deleted: assumed that it implements

Deleted: <#>11.8.7 The in operator¶
The production

Deleted: is evaluated as follows:

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: of calling the [[HasProperty]] internal method o

Deleted: with argument

Deleted: ToString

Formatted: Note

Deleted:
EqualityExpression [no *LineTerminator* here] **is** *RelationalExpression*
EqualityExpression [no *LineTerminator* here] **isnt** *RelationalExpression*

Deleted:
EqualityExpression [no *LineTerminator* here] **is** *RelationalExpression*
EqualityExpression [no *LineTerminator* here] **isnt** *RelationalExpression*

Deleted: Semantics¶
The result of evaluating an equality operator is always type Boolean, reflecting whether the relationship name by the operator holds between its two operands.¶

Deleted: evaluated in

Deleted: manner

Deleted: evaluated instead

Deleted:
EqualityExpression [no *LineTerminator* here] **is** *RelationalExpression*
EqualityExpression [no *LineTerminator* here] **isnt** *RelationalExpression*

11.9.1 Runtime Semantics

Runtime Semantics: The Abstract Equality Comparison

Deleted: Algorithm

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

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

Deleted: <#>ReturnIfAbrupt(*x*).If *x* is an abru completion, return *x*.¶
<#>ReturnIfAbrupt(*y*).If *y* is an abrupt completion, return *y*.¶

Deleted: s

Deleted: e

Deleted: c

Deleted: algorithm

Deleted: <#> If Type(*x*) is Undefined, return **true**.¶
<#>If Type(*x*) is Null, return **true**.¶
<#> If Type(*x*) is Number, then¶
<#>If *x* is **NaN**, return **false**.¶
<#>If *y* is **NaN**, return **false**.¶
<#>If *x* is the same Number value as *y*, return **true**.¶
<#>If *x* is **+0** and *y* is **−0**, return **true**.¶
<#>If *x* is **−0** and *y* is **+0**, return **true**.¶
<#>Return **false**.¶
<#>If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters i corresponding positions). Otherwise, return **false**.¶
<#>If Type(*x*) is Boolean, return **true** if *x* ar are both **true** or both **false**. Otherwise, return **false**.¶
<#>Return **true** if *x* and *y* refer toare the san objectObject value. Otherwise, return **false**.¶

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 normalised form.

Runtime Semantics: The Strict Equality Comparison

Deleted: Algorithm

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

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is Number, then
   a. If *x* is **NaN**, return **false**.
   b. If *y* is **NaN**, return **false**.
   c. If *x* is the same Number value as *y*, return **true**.
   d. If *x* is **+0** and *y* is **−0**, return **true**.
   e. If *x* is **−0** and *y* is **+0**, return **true**.
   f. Return **false**.
5. If Type(*x*) is String, then

Deleted: <#>ReturnIfAbrupt(*x*).If *x* is an abru completion, return *x*.¶
<#>ReturnIfAbrupt(*y*).If *y* is an abrupt completion, return *y*.¶

a. If *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions), return **true**.

b. Else, return **false**.

6. If Type(*x*) is Boolean, then

a. If *x* and *y* are both **true** or both **false**, return **true**.

b. Else, return **false**.

7. If *x* and *y* are the same Object value, return **true**.
8. Return **false**.

NOTE This algorithm differs from the SameValue Algorithm (9.12) in its treatment of signed zeroes and NaNs.

Runtime Semantics: Evaluation

EqualityExpression : *EqualityExpression* **==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of performing Abstract Equality Comparison *rval* == *lval*.

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

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

EqualityExpression **:** *EqualityExpression* **===** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of performing the Strict Equality Comparison *rval* === *lval*.

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

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *r* be the result of performing Strict Equality Comparison *rval* === *lval*.
8. If *r* is **true**, return **false**. Otherwise, return **true**.

Deleted: return **true** i
Deleted: ; otherwise
Deleted: return **true** i
Deleted: ;
Deleted: otherwise
Deleted: Return **true** i
Deleted: refer to
Deleted: object
Deleted: .
Deleted: Otherwise, r
Deleted: <#>**11.9.1 The Equals Operator (==)**¶ The production
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return *lval*.
Deleted: abstract
Deleted: equality
Deleted: comparison
Deleted: algorithm
Deleted: (see 11.9.3).
Deleted: <#>**11.9.2 The Does-not-equals Operator (!=)**¶
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return *lval*.
Deleted: If *rval* is an abrupt completion, return *rval*.
Deleted: abstract
Deleted: equality
Deleted: comparison
Deleted: algorithm
Deleted: (see 11.9.3).
Deleted: <#>**11.9.4 The Strict Equals Operator**
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return *lval*.
Deleted: strict
Deleted: equality
Deleted: comparison
Deleted: algorithm
Deleted: (See 11.9.6)
Deleted: <#>**11.9.5 The Strict Does-not-equal**
Deleted: is evaluated as follows:
Deleted: If *lval* is an abrupt completion, return *lval*.
Deleted: strict
Deleted: equality
Deleted: comparison
Deleted: algorithm
Deleted: (See 11.9.6)
Deleted: <#>ReturnIfAbrupt(*r*).¶

11.10 Binary Bitwise Operators

Syntax

BitwiseANDExpression **:**
  EqualityExpression
  BitwiseANDExpression **&** *EqualityExpression*

BitwiseANDExpressionNoIn **:**
  EqualityExpressionNoIn
  BitwiseANDExpressionNoIn **&** *EqualityExpressionNoIn*

BitwiseXORExpression **:**
  BitwiseANDExpression
  BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseXORExpressionNoIn **:**
  BitwiseANDExpressionNoIn
  BitwiseXORExpressionNoIn **^** *BitwiseANDExpressionNoIn*

BitwiseORExpression **:**
  BitwiseXORExpression
  BitwiseORExpression **|** *BitwiseXORExpression*

BitwiseORExpressionNoIn **:**
  BitwiseXORExpressionNoIn
  BitwiseORExpressionNoIn **|** *BitwiseXORExpressionNoIn*

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

Runtime Semantics: Evaluation

The production *A* **:** *A* @ *B*, where @ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Let *lref* be the result of evaluating *A*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *B*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToInt32(*rval*).
10. ReturnIfAbrupt(*rnum*).
11. Return the result of applying the bitwise operator @ to *lnum* and *rnum*. The result is a signed 32 bit integer.

11.11 Binary Logical Operators

Syntax

LogicalANDExpression **:**
  BitwiseORExpression
  LogicalANDExpression **&&** *BitwiseORExpression*

Deleted: *EqualityExpression* **:** *EqualityExpress* [no *LineTerminator* here] `is` *RelationalExpressi*
<#>Let *lref* be the result of evaluating *EqualityExpression*.¶
<#>Let *lval* be GetValue(*lref*).¶
<#>ReturnIfAbrupt(*lval*).If *lval* is an abrupt completion, return *lval*.¶
<#>Let *rref* be the result of evaluating *RelationalExpression*.¶
<#>Let *rval* be GetValue(*rref*).¶
<#>If *rval* is an abrupt completion, return *rval*.
<#>Return the result of performing SavmeValue(*rval*, *lval*). ¶
EqualityExpression **:** *EqualityExpression* [no *LineTerminator* here] `isnt` *RelationalExpressic*
<#>Let *lref* be the result of evaluating *EqualityExpression*.¶
<#>Let *lval* be GetValue(*lref*).¶
<#>ReturnIfAbrupt(*lval*).If *lval* is an abrupt completion, return *lval*.¶
<#>Let *rref* be the result of evaluating *RelationalExpression*.¶
<#>Let *rval* be GetValue(*rref*).¶
<#>Let *r* be the result of performing SavmeValue(*rval*, *lval*). ¶
<#>ReturnIfAbrupt(*r*).¶
<#>If *r* is **true**, return **false**. Otherwise, return **true**.¶

Deleted: Semantics¶

Deleted: If *lval* is an abrupt completion, returr *lval*.

Deleted: If *rval* is an abrupt completion, returr *rval*.

Deleted: If *lnum* is an abrupt completion, retu *lnum*.

Deleted: If *rnum* is an abrupt completion, retu *rnum*.

LogicalANDExpressionNoIn **:**
BitwiseORExpressionNoIn
LogicalANDExpressionNoIn **&&** *BitwiseORExpressionNoIn*

LogicalORExpression **:**
LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

LogicalORExpressionNoIn **:**
LogicalANDExpressionNoIn
LogicalORExpressionNoIn **||** *LogicalANDExpressionNoIn*

The semantics of the *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that the contained *LogicalANDExpressionNoIn*, *BitwiseORExpressionNoIn* and *LogicalORExpressionNoIn* are used in place of the contained *LogicalANDExpression*, *BitwiseORExpression* and *LogicalORExpression*, respectively.

Deleted: evaluated in

Deleted: evaluated instead

NOTE The value produced by a `&&` or `||` operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

Runtime Semantics: Evaluation

LogicalANDExpression **:** *LogicalANDExpression* **&&** *BitwiseORExpression*

Deleted: Semantics¶
The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalANDExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *lbool* be ToBoolean(*lval*).
4. ReturnIfAbrupt(*lbool*).
5. If *lbool* is **false**, return *lval*.
6. Let *rref* be the result of evaluating *BitwiseORExpression*.
7. Return GetValue(*rref*).

Deleted: *lval*

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: *lval*ToBoolean(*lval*)

LogicalORExpression **:** *LogicalORExpression* **||** *LogicalANDExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *lbool* be ToBoolean(*lval*).
4. ReturnIfAbrupt(*lbool*).
5. If *lbool* is **true**, return *lval*.
6. Let *rref* be the result of evaluating *LogicalANDExpression*.
7. Return GetValue(*rref*).

Deleted: *lval*

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: *lval*ToBoolean(*lval*)

11.12 Conditional Operator (? :)

Deleted: <#>The *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are evaluate in the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that t contained *LogicalANDExpressionNoIn*, *BitwiseORExpressionNoIn* and *LogicalORExpressionNoIn* are evaluated instead of the contained *LogicalANDExpression*, *BitwiseORExpression* and *LogicalORExpression*, respectively.¶
<#>NOTE The value produced by a && or || operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.¶

Syntax

ConditionalExpression **:**
LogicalORExpression
LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

ConditionalExpressionNoIn **:**
LogicalORExpressionNoIn
LogicalORExpressionNoIn **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

The semantics of the *ConditionalExpressionNoIn* production is the same as the *ConditionalExpression* production except that the contained *LogicalORExpressionNoIn*, *AssignmentExpression* and *AssignmentExpressionNoIn* are used in place of the contained *LogicalORExpression*, first *AssignmentExpression* and second *AssignmentExpression*, respectively.

NOTE The grammar for a ConditionalExpression in ECMAScript is a little bit 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.

Static Semantics: IsValidSimpleAssignmentTarget

ConditionalExpression **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*

1. Return **false**.

Runtime Semantics: Evaluation

ConditionalExpression **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be ToBoolean(GetValue(*lref*)).
3. ReturnIfAbrupt(*lval*).
4. If *lval* is **true**, then
   a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
   b. Return GetValue(*trueRef*).
5. Else
   a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
   b. Return GetValue(*falseRef*).

11.13 Assignment Operators

Syntax

AssignmentExpression **:**
ConditionalExpression
YieldExpression
ArrowFunction
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn **:**
ConditionalExpressionNoIn
YieldExpression
ArrowFunction
LeftHandSideExpression **=** *AssignmentExpressionNoIn*
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

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

The semantics of the *AssignmentExpressionNoIn* productions are the same manner as the *AssignmentExpression* productions except that the contained *ConditionalExpressionNoIn* and *AssignmentExpressionNoIn* are used in place of the contained *ConditionalExpression* and *AssignmentExpression*, respectively.

11.13.1 Static Semantics

Static Semantics: Early Errors

Deleted: evaluated in

Deleted: manner

Deleted: evaluated instead

Deleted: Semantics¶
The production

Deleted: is evaluated as follows:

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: ToBoolean(GetValue(*lref*))

Deleted: <#>Supplemental Syntax¶
<#>In certain circumstances when processing the production *AssignmentExpression* : *LeftHandSideExpression* = *AssignmentExpression* the following grammar is used to refine the interpretation of *LeftHandSideExpression*.
<#>AssignmentPattern :¶
<#>ObjectAssignmentPattern
ArrayAssignmentPattern ¶
<#>ObjectAssignmentPattern :¶
<#>{ }
{ AssignmentPropertyList }
{ AssignmentPropertyList , }¶
<#>ArrayAssignmentPattern :¶
<#>[$Elision_{opt}$ $AssignmentRestElement_{opt}$]
[AssignmentElementList , $Elision_{opt}$ $AssignmentRestElement_{opt}$]¶
<#>AssignmentPropertyList :¶
<#>AssignmentProperty
AssignmentPropertyList , AssignmentProperty¶
<#>AssignmentElementList :¶
<#>$Elision_{opt}$ AssignmentElement
AssignmentElementList , $Elision_{opt}$ AssignmentElement¶
<#>AssignmentProperty :¶
<#>Identifier
PropertyName : LeftHandSideExpression¶
<#>AssignmentElement :¶
<#> LeftHandSideExpression¶
<#>AssignmentRestElement :¶
<#> ... LeftHandSideExpression¶

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

- It is a Syntax Error if *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* cannot be parsed with no tokens left over using *AssignmentPattern* as the goal symbol.
- If *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* can be parsed with no tokens left over using *AssignmentPattern* as the goal symbol then the following rules are not applied. Instead, the Early Error rules for *AssignmentPattern* are used.
- It is a Syntax Error if *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is an early Reference Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.

AssignmentExpression **:** *LeftHandSideExpression AssignmentOperator AssignmentExpression*

- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is an early Reference Error if IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.

Static Semantics: IsValidSimpleAssignmentTarget

AssignmentExpression **:**
    YieldExpression
    ArrowFunction
    LeftHandSideExpression **=** *AssignmentExpression*
    LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Return **false**.

11.13.2 Runtime Semantics

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. Let *status* be PutValue(*lref*, *rval*).
   f. ReturnIfAbrupt(*status*).
   g. Return *rval*.
2. Let *AssignmentPattern* be the parse of the source code corresponding to *LeftHandSideExpression* using *AssignmentPattern* as the goal symbol.
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Let *rval* be ToObject(GetValue(*rref*)).
5. ReturnIfAbrupt(*rval*).
6. Let *status* be the result of performing Destructuring Assignment Evaluation of *AssignmentPattern* using *rval* as the argument.
7. ReturnIfAbrupt(*status*).
8. Return *rval*.

AssignmentExpression **:** *LeftHandSideExpression AssignmentOperator AssignmentExpression*

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *lval* be GetValue(*lref*).

Deleted: *AssignmentExpression* **:**¶
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression AssignmentOperator AssignmentExpression¶
<#>Semantics¶
<#>11.13.1 Simple Assignment (=)¶
<#>The static semantics of *AssignmentExpression* **:** *LeftHandSideExpression* **=** *AssignmentExpression* are:¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in strict code and *LeftHandSideExpression* i… the *Identifier* **eval** or the *Identifier* **arguments**.¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in extended code and the *LeftHandSideExpression* is the *Identifier* **this** or the *Identifer* **super**.¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in extended code and the *LeftHandSideExpression* is a *Literal* or a *FunctionExpression*.¶
<#>It is a Syntax Error if the *AssignmentExpression* is contained in extended code and the *LeftHandSideExpression* is an *Identifier* that can be do… not statically determined to always resolves to a declarative environment record binding orand if the resolved binding is an immutable binding.¶
<#>It is a Syntax Error if the *LeftHandSideExpression* … *PrimaryExpression* : **(** *Expression* **)** and *Expression* derived a production that would produce a Syntax E… according to these rules. This rule is recursively applied.¶

Deleted: the

Deleted: <#>It is a Syntax Error if the *AssignmentExpression* is contained in extended code and IsInvalidAssignmentPattern of the *LeftHandSideExpress…* is an ***ObjectLiteral*** **or an** ***ArrayLiteral*** **and the source c… corresponding to** ***LeftHandSideExpression*** **cannot be parsed using** ***AssignmentPattern*** **as the goal symboltru…**

Deleted: The production

Deleted: is evaluated as follows:

Deleted: If *lref* is an abrupt completion, return *lref*.

Deleted: <#>Throw a **SyntaxError** exception if the following conditions are all true:¶
<#> Type(*lref*) is Reference is **true**¶
<#> IsStrictReference(*lref*) is **true**¶
<#> Type(GetBase(*lref*)) is Environment Record¶
<#> GetReferencedName(*lref*) is either **"eval"** or **"arguments"**¶

Deleted: Call

Deleted: If the result of

Deleted: <#>If this is not extended code, throw a **ReferenceError** exception.¶

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: P

Deleted: Evaluate

Deleted: *obj* parameter

Deleted: If *status* is an abrupt completion, return *rval*.

Deleted:
where *AssignmentOperator* is @= and @ represents o… of the operators indicated above

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

NOTE When an assignment occurs within strict mode code, it is an runtime error if *lref* in step 1.e of the first algorithm or step 9 of the second algorithm it is an unresolvable reference. If it is, a **ReferenceError** exception is thrown. The *LeftHandSide* also may not be a reference to a data property with the attribute value {[[Writable]]:**false**}, to an accessor property with the attribute value {[[Set]]:**undefined**}, nor to a non-existent property of an object for which the IsExtensible predicate returns the value **false**. In these cases a **TypeError** exception is thrown.

11.13.3 Destructuring Assignment

Supplemental Syntax

In certain circumstances when processing the production *AssignmentExpression* : *LeftHandSideExpression* = *AssignmentExpression* the following grammar is used to refine the interpretation of *LeftHandSideExpression*.

AssignmentPattern :
  ObjectAssignmentPattern
  ArrayAssignmentPattern

ObjectAssignmentPattern :
  { }
  { *AssignmentPropertyList* }
  { *AssignmentPropertyList* , }

ArrayAssignmentPattern :
  [$Elision_{opt}$ $AssignmentRestElement_{opt}$]
  [*AssignmentElementList*]
  [*AssignmentElementList* , $Elision_{opt}$ $AssignmentRestElement_{opt}$]

AssignmentPropertyList :
  AssignmentProperty
  AssignmentPropertyList , *AssignmentProperty*

AssignmentElementList :
  $Elision_{opt}$ *AssignmentElement*
  AssignmentElementList , $Elision_{opt}$ *AssignmentElement*

AssignmentProperty :
  Identifier $Initialiser_{opt}$
  PropertyName : *AssignmentElement*

AssignmentElement :
  DestructuringAssignmentTarget $Initialiser_{opt}$

AssignmentRestElement :
  ... *DestructuringAssignmentTarget*

DestructuringAssignmentTarget :
  LeftHandSideExpression

Deleted: If *lval* is an abrupt completion, return *lval*.

Deleted: If *rval* is an abrupt completion, return *rval*.

Deleted: <#>Throw a **SyntaxError** exception if the following conditions are all true:¶
<#> Type(*lref*) is Reference is **true**¶
<#> IsStrictReference(*lref*) is **true**¶
<#> Type(GetBase(*lref*)) is Environment Rec
<#> GetReferencedName(lref) is either `"eva` or `"arguments"`¶

Deleted: If the result of

Deleted: Call

Deleted: is an abrupt completion, return that result

Deleted: ¶

Formatted: Note

Deleted: *its LeftHandSide must not evaluate to*

Deleted: d

Deleted:

Deleted: must not be

Deleted: an

Deleted: does

Deleted: upon assignment

Deleted: whose

Deleted: where calling its

Deleted: [[

Deleted: Get

Deleted:]] internal property method

Deleted: has

Deleted: ¶

Deleted: *LeftHandSideExpression*

Deleted: *LeftHandSideExpression*

Deleted: ...

Deleted: *LeftHandSideExpression*

11.13.3.1 Static Semantics

Static Semantics: Early Errors

AssignmentProperty **:** *Identifier* *Initialiser*$_{opt}$

- It is a Syntax Error if *Identifier* is the *Identifier* **`eval`** or the *Identifier* **`arguments`**.
- It is a Syntax Error if *Identifier* does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.

AssignmentRestElement **:** **`...`** *DestructuringAssignmentTarget*

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

DestructuringAssignmentTarget **:** *LeftHandSideExpression*

- It is a Syntax Error *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 of *LeftHandSideExpression* is **false**.
- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if *LeftHandSideExpression* is the *Identifier* **`eval`** or the *Identifier* **`arguments`**.
- It is a Syntax Error if IsInvalidAssignmentPattern of *LeftHandSideExpression* is **true**.
- It is a Syntax Error if the *LeftHandSideExpression* is *CoverParenthesisedExpressionAndArrowParameterList* **:** **(** *Expression* **)** and *Expression* derives a production that would produce a Syntax Error according to these rules if that production is substituted for *LeftHandSideExpression*. This rule is recursively applied.

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

11.13.3.2 Runtime Semantics

Runtime Semantics: Destructuring Assignment Evaluation

with parameter *obj*

ObjectAssignmentPattern **:** **{** **}**
and
ArrayAssignmentPattern **:**
[ ]
[*Elision* **]**

1. Return NormalCompletion(empty).

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

1. Let *status* be the result of performing Destructuring Assignment Evaluation for *AssignmentPropertyList* using *obj* as the argument.
2. ReturnIfAbrupt(*status*).
3. Return the result of performing Destructuring Assignment Evaluation for *AssignmentProperty* using *obj* as the argument.

AssignmentProperty **:** *Identifier* *Initialiser*$_{opt}$

1. Let *P* be StringValue of *Identifier*.

Deleted: *PropertyName*
Deleted: :
Deleted: *LeftHandSideExpression*
Deleted: <#>It is a Syntax Error if *Identifier* is the *Identifier* **`this`** or the *Identifier* **`super`**.¶
Comment [AW80]: This part probably doesn't need t be here if 11.1.2 has this as a static semantic for extended code.
Deleted: *AssignmentElement* **:** *LeftHandSideExpression Initialiser*$_{opt}$¶
Comment [AW81]: This part probably doesn't need t be here if 11.1.2 has this as a static semantic for extended code.
Deleted: <#>It is a Syntax Error if *LeftHandSideExpression* is the *Identifier* **`this`** or the *Identifier* **`super`**.¶
<#>It is a Syntax Error if the *LeftHandSideExpression* a *Literal*, a *FunctionExpression* or a *ClassExpression*.¶
<#>It is a Syntax Error if the *LeftHandSideExpression* an *Identifier* that does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.¶
Deleted: It is a Syntax Error if the *LeftHandSideExpression* is an *ObjectLiteral* or an
Deleted: *z*
Deleted: *PrimaryExpression*
Deleted:
Deleted: d
Deleted: <#>*AssignmentRestElement* : ...
Deleted: The supplemental production
Deleted: The supplemental production
Deleted: ,the production *ArrayAssignmentPattern* : [
Deleted: *ObjectAssignmentPattern* :¶
Deleted: The supplemental production
Deleted: is evaluated with the parameter *obj* as
Deleted: Perform
Deleted: of
Deleted: Evaluate
Deleted: with
Deleted: using
Deleted: *obj* parameter
Deleted: If *status* is an abrupt completion, return
Deleted:
Deleted: Perform
Deleted: of Evaluate *AssignmentProperty*
Deleted: with
Deleted: using
Deleted: *obj*
Deleted: parameter
Deleted: The supplemental production
Deleted:

2. Let *v* be the result of calling Get(*obj*, *P*).
3. ReturnIfAbrupt(*v*).
4. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be ToObject(GetValue(*defaultValue*)).
5. ReturnIfAbrupt(*v*).
6. Let *lref* be the result of performing Identifier Resolution(10.3.1) with the *IdentifierName* corresponding to *Identifier*.
7. Return PutValue(*lref*,*v*).

AssignmentProperty **:** *PropertyName* **:** *AssignmentElement*

1. Let *name* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*Name*).
3. Return the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.

ArrayAssignmentPattern **: [** *Elision*$_{opt}$ *AssignmentRestElement* **]**

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentRestElement* with *obj* and *skip* as the arguments.

ArrayAssignmentPattern **: [** *AssignmentElementList* **]**

1. Return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* and 0 as the arguments.

ArrayAssignmentPattern **: [** *AssignmentElementList* **,** *Elision*$_{opt}$ *AssignmentRestElement*$_{opt}$ **]**

1. Let *lastIndex* be the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* and 0 as the arguments.
2. ReturnIfAbrupt(*lastIndex*).
3. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. If *AssignmentRestElement* is present, then return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentRestElement* with *obj* and *lastIndex*+*skip* as the arguments.
5. Return *lastIndex*.

Runtime Semantics: Indexed Destructuring Assignment Evaluation

with parameters *obj* and *index*

AssignmentElementList **:** *Elision*$_{opt}$ *AssignmentElement*

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *name* be ToString(*index*+*skip*).
3. Let *status* be the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.
4. ReturnIfAbrupt(*status*).
5. Return *index*+*skip*+1.

AssignmentElementList **:** *AssignmentElementList* **,** *Elision*$_{opt}$ *AssignmentElement*

1. Let *listNext* be the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* as the *obj* parameter and *index* as the *index* parameter
2. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
3. ReturnIfAbrupt(*listNext*).
4. Let *name* be ToString(*listNext*+*skip*).

Deleted: <#>Let *exists* be the result of calling
Deleted: the [[…et(]] internal method of …*bj*
Deleted: <#>Else ¶
Deleted: *v*
Deleted: Else,
Deleted: <#> Let *v* be **undefined**.¶
Deleted: using
Deleted: Call
Deleted: ¶
Deleted: The static semantics of the
Deleted: *LeftHandSideExpression*
Deleted: is evaluated with the parameter
Deleted: the result of evaluating
Deleted: PropName of
Deleted:
Deleted: Indexed
Deleted: <#>Let *v* be the result of calling the
Deleted: The supplemental production
Deleted:
Deleted: is evaluated with the parameter
Deleted: If *Elision* is present, then let *skip* be
Deleted: Perform
Deleted: Evaluate …*ssignmentRestElement* w
Deleted: 0
Deleted: *index* …rgumentsparameter…¶
Deleted: <#>Perform Indexed Destructuring
Deleted: The supplemental production
Deleted: *AssignmentPropertyList*
Deleted: .
Deleted: <#>is evaluated with the paramet
Deleted: *index*
Deleted: evaluating …*ssignmentElementList*
Deleted: If *lastIndex* is an abrupt completion,
Deleted: If *Elision* is present, then let *skip* be
Deleted: perform
Deleted: evaluate …*ssignmentRestElement* w
Deleted: *i*
Deleted: *index* …rgumentsparameter…¶
Deleted: The supplemental production
Deleted: is evaluated with the parameters
Deleted: If *Elision* is present, then let *skip* be
Deleted: Perform
Deleted: Indexed
Deleted: Evaluate
Deleted: Let *index* be the result of evaluating
Deleted: using …*bj* as the *obj* parameter
Deleted: *index*+*skip*
Deleted: *index* …rgumentsparameter
Deleted: If *status* is an abrupt completion, retu
Deleted: +*skip*
Deleted: ¶
Deleted: The supplemental production
Deleted: is evaluated with the parameters
Deleted: *e*…*istNextnd*
Deleted: evaluating

5. Let *status* be the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.
6. ReturnIfAbrupt(*status*).
7. Return *listNext*+*skip*+1.

AssignmentRestElement **:** ... *DestructuringAssignmentTarget*

1. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
2. ReturnIfAbrupt(*lref*).
3. Let *lenVal* be the result of Get(*obj*, `"length"`).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
7. Let *n*=0;
8. Repeat, while *index* < *len*
    a. Let *P* be ToString(*index*).
    b. Let *exists* be the result of HasProperty(*obj*, *P*).
    c. ReturnIfAbrupt(*exists*).
    d. If *exists* is **true**, then
        i. Let *v* be the result of Get(*obj*, ToString(*index*)).
        ii. ReturnIfAbrupt(*len*).
        iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Let *n* = *n*+1.
    f. Let *index* = *index*+1.
9. Return PutValue(*lref*,*A*).

Runtime Semantics: Keyed Destructuring Assignment Evaluation

with parameters *obj* and *propertyName*

AssignmentElement **:** *DestructuringAssignmentTarget* *Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
    a. Let *defaultValue* be the result of evaluating *Initialiser*.
    b. Let *v* be GetValue(*defaultValue*)
4. If *DestructuringAssignmentTarget* is an *ObjectLiteral* or an *ArrayLiteral* then
    a. Let *AssignmentPattern* be the parse of the source code corresponding to *DestructuringAssignmentTarget* using *AssignmentPattern* as the goal symbol
    b. Let *vObj* be ToObject(*v*).
    c. ReturnIfAbrupt(*vObj*).
    d. Return the result of performing Destructuring Assignment Evaluation of *AssignmentPattern* with *vObj* as the argument.
5. ReturnIfAbrupt(*v*).
6. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
7. Return PutValue(*lref*,*v*).

11.14 Comma Operator (,)

Syntax

Expression **:**
  AssignmentExpression
  Expression **,** *AssignmentExpression*

ExpressionNoIn **:**
  AssignmentExpressionNoIn
  ExpressionNoIn **,** *AssignmentExpressionNoIn*

Deleted: Let *index* be the result of evaluating

Deleted: Perform

Deleted: Indexed

Deleted: Evaluate ...*ssignmentElement* with using ...*bj*

Deleted: *e*

Deleted: *listNextnd+ski*

Deleted: *p*

Deleted: *index* ...rgumentsparameter

Deleted: If *status* is an abrupt completion, return *status*

Deleted: *i*...*istNextndex*

Deleted: ¶

Deleted: The supplemental production *AssignmentElement* **:** *LeftHandSideExpression Initialiser*$_{opt}$¶ is evaluated with the parameters *obj* and *index* as follows:¶
<#>Let *name* be ToString(*index*).¶
<#>Let *exists* be the result of calling the [[HasProperty]] internal method of *obj* with argument *name*. ¶
<#>If *exists* is **true**, then¶
<#>Let *v* be the result of calling the [[Get]] internal metho of *obj* with argument *name*.¶
<#>Else ¶
<#>If *Initialiser*$_{opt}$ is present, then¶
<#>Let *v* be the result of evaluating *Initialiser*.¶
<#>Else,¶
<#> Let *v* be **undefined**.¶

Deleted: The supplemental production

Deleted: ...

Deleted: *LeftHandSideExpression*

Deleted: is evaluated with the parameters *obj* and

Deleted: *LeftHandSideExpression*

Deleted: If *lref* is an abrupt completion,

Deleted: calling the [[...et(]] internal method of ...*bj* wit

Deleted: t

Deleted: If *len* is an abrupt completion, return *len*

Deleted: ion

Deleted: (15.4)

Deleted: a new array object created as if by the expressio

Deleted: calling the [[HasProperty]] internal method o

Deleted: calling the [[...et]]... internal method of ...

Deleted: If *v* is an abrupt completion, return *v*.

Deleted: ,...and Property Descriptor {[[Value]]: *v*,

Deleted: Call

Deleted: ¶

Deleted: *LeftHandSideExpression*

Deleted: calling the [[...et]]... internal method of ...*bj* w

Deleted: *v*

Deleted: *LeftHandSideExpression*

Deleted: *LeftHandSideExpression*

Deleted: *LeftHandSideExpression*

The semantics of the *ExpressionNoIn* production is the same manner as the *Expression* production except that the contained *ExpressionNoIn* and *AssignmentExpressionNoIn* are used in place of the contained *Expression* and *AssignmentExpression*, respectively.

Static Semantics: IsValidSimpleAssignmentTarget

Expression **:** *Expression* **,** *AssignmentExpression*

1. Return **false**.

Runtime Semantics: Evaluation

Expression **:** *Expression* **,** *AssignmentExpression*

1. Let *lref* be the result of evaluating *Expression*.
2. ReturnIfAbrupt(GetValue(*lref*)).
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Return GetValue(*rref*).

NOTE GetValue must be called even though its value is not used because it may have observable side-effects.

12 Statements and Declarations

Syntax

Statement **:**
BlockStatement
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
BreakableStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
ThrowStatement
TryStatement
DebuggerStatement

Declaration **:**
FunctionDeclaration
GeneratorDeclaration
ClassDeclaration
LexicalDeclaration

BreakableStatement **:**
IterationStatement
SwitchStatement

Static Semantics

Static Semantics: VarDeclaredNames

Deleted: evaluated in

Deleted: evaluated instead

Deleted: Semantics¶
The production

Deleted: is evaluated as follows:

Deleted: Call

Deleted: If the result of

Deleted:

Deleted: is an abrupt completion, return that result

Deleted: .

Deleted: *IterationStatement*

Deleted: *SwitchStatement*

Deleted: *LetDeclaration* *Const*

Deleted: A *Statement* can be part of a *LabelledStatement*, which itself can be part of a *LabelledStatement*, and so on. The labels introduced this way are collectively referred to as the "current label set" when describing the semantics of individual statements. A *LabelledStatement* has no semantic meaning other than the introduction of a label to a *label set*. The label set of an *IterationStatement* or a *SwitchStatement* initially contains the single element **empty**. The label set of any other statement is initially empty.¶

Deleted: Static Semantics: BoundNames
The BoundNames of the *Declaration* production are determined as follows:¶
<#>Return the BoundNames of the single non-terminal symbol that is the right hand side of the production.¶

Statement **:**
EmptyStatement
ExpressionStatement
ContinueStatement
BreakStatement
ReturnStatement
ThrowStatement
DebuggerStatement

1. Return a new empty List.

Runtime Semantics

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

BreakableStatement **:** *IterationStatement*

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

BreakableStatement **:** *SwitchStatement*

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

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

Runtime Semantics: Evaluation

BreakableStatement **:**
IterationStatement
SwitchStatement

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

12.1 Block

Syntax

BlockStatement **:**
Block

Block **:**
{ *StatementList*$_{opt}$ **}**

StatementList **:**
StatementListItem
StatementList StatementListItem

Deleted: The VarDeclaredNames of the productions:
Statement **:** *EmptyStatement*
Statement **:** *ExpressionStatement*
Statement **:** *ContinueStatement*
Statement **:** *BreakStatement*
Statement **:** *ReturnStatement*
Statement **:** *ThrowStatement*
Statement **:** *DebuggerStatement*
is determined as follows:¶

Deleted: Statement :¶
BlockStatement
VariableStatement
IterationStatement
WithStatement
LabelledStatement
SwitchStatement
TryStatement ¶
The VarDeclaredNames of the productions:
Statement : BlockStatement
Statement : IfStatement
Statement : IterationStatement
Statement : WithStatement
Statement : LabelledStatement
Statement : SwitchStatement
Statement : TryStatement
is determined as follows:¶
Return the VarDeclaredNames of the single non-terminal symbol that is the right hand side of the production.¶
The VarDeclaredNames of the production Statement : VariableStatement is determined as follows:¶
Return the BoundNames of VariableStatement.¶

Deleted: L

Deleted: L

Deleted: s

Comment [AWB 382]: Need to have an evlaution semantics in section 13 for *FunctionDeclaration*

Deleted: The *Statement* productions are all evaluated as follows¶
<#>Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.¶
The *Declaration* productions are all evaluated as as follows¶
<#>Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.¶
NOTE The result of evaluating a *Statement* or *Declaration* is always a Completion value.¶

Deleted: <#>The BoundNames of the *Declaration* productions are determined as follows:¶
<#>Return the BoundNames of the single non-terminal symbol that is the right hand side of the production.¶
<#>The *Declaration* productions are all evaluated as as follows¶
<#>Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.¶

Deleted: <#>NOTE Several widely used implementations of ECMAScript are known to support the use of FunctionDeclaration as a Statement. However there are significant and irreconcilable variations among the implementations in the semantics applied to such *FunctionDeclarations*. Because of these irreconcilable differences, the use of a *FunctionDeclaration* as a *Statement* results in code that is not reliably portable among implementations. It is recommended that

StatementListItem **:**
Statement
Declaration

12.1.1.1 Static Semantics

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*.

Static Semantics: LexicalDeclarations

StatementList **:** *StatementList StatementListItem*

1. Let *declarations* be LexicalDeclarations of *StatementList*.
2. Append to *declarations* the elements of the LexicalDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. Return a new List containing *Declaration*.

Static Semantics: LexicallyDeclaredNames

Block **: { }**

1. Return a new empty List.

StatementList **:** *StatementList StatementListItem*

1. Let *names* be LexicallyDeclaredNames of *StatementList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. Return the BoundNames of *Declaration*.

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*

Deleted: <#>It is a Syntax Error if *StatementList* includes a *StatementListItem* **:** *Declaration* production whose *Declaration* is *Declaration* **:** *FunctionDeclaration* production and the source code matching this *Block* production is not contained in extended co

Deleted: i

Deleted: *StatementList* **:** *StatementListItem* ¶
<#>Return the LexicalDeclarations of *StatementListItem*.¶

Deleted: *names*

Deleted: return

Deleted: The VarDeclaredNames of the production *BlockStatement* **:** *Block* is determin as follows:¶
<#>Return the VarDeclaredNames of *Block*.¶
The production *BlockStatement* **:** *Block* is evaluated as follows:¶
<#>Return the result of evaluating *Block*.¶

Deleted: The LexicallyDeclaredNames of th production

Deleted: is determined as follows:

Deleted: *Block* **: {** *StatementList* **}** ¶
<#>Return the LexicallyDeclaredNames of *StatementListItem*.¶
StatementList **:** *StatementListItem* ¶
<#>Return the LexicallyDeclaredNames of *StatementListItem*.¶

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return a new empty List.
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.

Deleted: ..

Static Semantics: TopLevelLexicallyScopedDeclarations

StatementList **:** *StatementList StatementListItem*

1. Let *declarations* be TopLevelLexicallyScopedDeclarations of *StatementList*.
2. Append to *declarations* the elements of the TopLevelLexicallyScopedDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

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

Static Semantics: TopLevelVarDeclaredNames

StatementList **:** *StatementList StatementListItem*

1. Let *names* be TopLevelVarDeclaredNames of *StatementList*.
2. Append to *names* the elements of the TopLevelVarDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return the LexicallyDeclaredNames of *Declaration*.
2. Return a new empty List.

StatementListItem **:** *Statement*

1. Return VarDeclaredNames of *Statement*.

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

Static Semantics: TopLevelVarScopedDeclarations

StatementList **:** *StatementList StatementListItem*

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

StatementListItem **:** *Statement*

1. If *Statement* is *Statement* **:** *VariableStatement*, then return a new List containing *VariableStatement*.
2. Return a new empty List.

StatementListItem **:** *Declaration*

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

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*

2. Return a new empty List.

12.1.1.2 Runtime Semantics

Runtime Semantics: Evaluation

Block **: { }**

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

Block **: {** *StatementList* **}**

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *blockEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
3. Perform Block Declaration Instantiation using *StatementList* and *blockEnv*.
4. Set the running execution context's LexicalEnvironment to *blockEnv*.
5. Let *blockValue* be the result of evaluating *StatementList*.
6. Set the running execution context's LexicalEnvironment to *oldEnv*.
7. If *blockValue*.[[type]] is normal and *blockValue*.[[value]] is empty, then
   a. Return NormalCompletion(**undefined**).
8. Return *blockValue*.

NOTE No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

StatementList **:** *StatementList StatementListItem*

1. Let *sl* be the result of evaluating *StatementList*.
2. ReturnIfAbrupt(*sl*).
3. Let *s* be the result of evaluating *StatementListItem*.
4. If *s*.[[type]] is throw, return *s*.
5. If *s*.[[value]] is empty, let *V* = *sl*.[[value]], otherwise let *V* = *s*.[[value]].
6. Return Completion {[[type]]: *s*.[[type]], [[value]]: *V*, [[target]]: *s*.[[target]]}.

NOTE Steps 4 and 5 of the above algorithm ensure that the value of a *StatementList* is the value of the last value producing *Statement* in the *StatementList*. For example, the following calls to the **eval** function all return the value 1:

```
eval("1;;;;;")
eval("1;{}")
eval("1;var a;")
```

Deleted: *BlockStatement* **:** *Block* ¶
<#>Return the VarDeclaredNames of *Block*.¶

Deleted: The VarDeclaredNames of the production

Deleted: is determined as follows:

Deleted: *Block* **: {** *StatementList* **}** ¶
<#>Return the VarDeclaredNames of *StatementList*.¶
StatementList **:** *StatementListItem* ¶
<#>Return the VarDeclaredNames of *StatementListItem*.¶

Deleted: *StatementListItem* **:** *Statement* ¶
<#>Return the VarDeclaredNames of *Statemen*

Deleted: *BlockStatement* **:** *Block* ¶

Deleted: The production

Deleted: is evaluated as follows:

Comment [AWB1383]: Breaking change:

Comment [AWB1384]: TODO, need to ve

Deleted: normal,

Deleted: empty

Deleted: , **empty**

Deleted: The static semantics of the

Deleted: is evaluated as follows:

Deleted: running

Deleted: current

Deleted: For each element *param*, of *formal*

Deleted: running

Deleted: current

Deleted: running

Deleted: current

Comment [AWB1385]: Breaking change:

Comment [AW86]: If evaluation of code st

Deleted: The LexicallyDeclaredNames of th

Deleted: The production *StatementList* **:**

Deleted: The LexicallyDeclaredNames of th

Deleted: is evaluated as follows:

Deleted: If *sl* is an abrupt completion, return s

Deleted: *an exception was thrown, return (*

Deleted: *V*, empty)

Deleted: where

Deleted: *V* is the exception. (Execution now

Deleted: value

Deleted: (

Deleted: type

Deleted:).

Deleted: 5

Deleted: 6

Comment [AWB1387]: ISSUE: above

12.2 Declarations and the Variable Statement

12.2.1 Let and Const Declarations

NOTE A `let` and `const` declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initialiser* is assigned the value of its *Initialiser's AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a `let` declaration does not have an *Initialiser* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

Syntax

LexicalDeclaration **:**
  LetOrConst BindingList **;**

LexicalDeclarationNoIn **:**
  LetOrConst BindingListNoIn

LetOrConst **:**
  `let`
  `const`

BindingList **:**
  LexicalBinding
  BindingList **,** *LexicalBinding*

BindingListNoIn **:**
  LexicalBindingNoIn
  BindingListNoIn **,** *LexicalBindingNoIn*

LexicalBinding **:**
  BindingIdentifier Initialiser$_{opt}$
  BindingPattern Initialiser

LexicalBindingNoIn **:**
  BindingIdentifier InitialiserNoIn$_{opt}$
  BindingPattern InitialiserNoIn

BindingIdentifier **:**
  Identifier

InitialiserNoIn **:**
  = *AssignmentExpressionNoIn*

The semantics of the *LexicalDeclarationNoIn*, *BindingListNoIn*, *LexicalBindingNoIn* and *InitialiserNoIn* productions are the same as the *LexicalDeclaration*, *BindingList*, *LexicalBinding* and *Initialiser* productions except that the contained *BindingListNoIn*, *LexicalBindingNoIn*, *InitialiserNoIn* and *AssignmentExpressionNoIn* are used in place of the contained *BindingList*, *LexicalBinding*, *Initialiser* and *AssignmentExpression*, respectively.

12.2.1.1 Static Semantics

Static Semantics: Early Errors

LexicalBinding **:** *BindingIdentifier*

- It is a Syntax Error if IsConstantDeclaration of the *LexicalDeclaration* containing this production is **true**.

BindingIdentifier **:** *Identifier*

Deleted: <#>The LexicallyDeclaredNames and the LexicalDeclarations of the production *StatementListItem* **:** *Statement* are determined as follows:¶
<#>Return a new empty List.¶

Deleted: <#>The production *StatementListItem* **:**

Deleted: <#>The LexicallyDeclaredNames of th

Deleted: <#>The production *StatementListItem* **:**

Deleted: s

Deleted: current

Deleted: Let

Deleted: *Let*

Deleted: executed

Deleted: *LetBinding*

Deleted: *t*

Deleted: executed

Deleted: *LetBinding*

Deleted: an

Deleted: *t*

Deleted: executed

Deleted: *Let*

Deleted: `let` *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Initialiser* **:**¶

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: *Let*

Deleted: <#>NOTE A `let` declaration defines

Deleted: if the *BindingIdentifier*

Comment [AWB888]: Consider disallowing undefine

Deleted: Semantics¶

Deleted: The static semantics of the production

- It is a Syntax Error if the *BindingIdentifier* is contained in strict code and if the *Identifier* is **`eval`** or **`arguments`**.

Static Semantics: BoundNames

LexicalDeclaration **:** *LetOrConst BindingList* **;**

1. Return the BoundNames of *BindingList*.

BindingList **:** *BindingList* **,** *LexicalBinding*

1. Let *names* be the BoundNames of *BindingList*.
2. Append to *names* the elements of the BoundNames of *LexicalBinding*.
3. Return *names*.

LexicalBinding **:** *BindingIdentifier Initialiser*$_{opt}$

1. Return the BoundNames of *BindingIdentifier*.

LexicalBinding **:** *BindingPattern Initialiser*

1. Return the BoundNames of *BindingPattern*.

BindingIdentifier **:** *Identifier*

1. Return a new List containing the StringValue of *Identifier*.

Static Semantics: IsConstantDeclaration

LexicalDeclaration **:** *LetOrConst BindingList* **;**

1. Return IsConstantDeclaration of *LetOrConst*.

LetOrConst **:** **`let`**

1. Return **false**.

LetOrConst **:** **`const`**

1. Return **true**.

12.2.1.2 Runtime Semantics

Runtime Semantics: Binding Initialisation

With arguments *value* and *environment*.

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

BindingIdentifier **:** *Identifier*

1. If *environment* is not **undefined**, then
   a. Let *name* be StringValue of *Identifier*.
   b. Let *env* be the environment record component of *environment*.
   c. Call the InitialiseBinding concrete method of *env* passing *name* and *value* as the arguments.
   d. Return NormalCompletion(**undefined**).
2. Else

Deleted: Bound Names
Deleted: *The BoundNames of the production*
Deleted: *Let*
Deleted: ***let*** *LetBindingList*
Deleted: is determined as follows:
Deleted: Bound Names
Deleted: *Let*
Deleted: *LetBindingList* : *LetLexicalBinding* ¶ <#>Return the Bound Names of *LetLexicalBinding*.¶
Deleted: *Let*
Deleted: *Let*
Deleted: *Let*
Deleted: N
Deleted:
Deleted: *Let*
Deleted: of
Deleted: Bound Names
Deleted: *Let*
Deleted: *Let*
Deleted: Bound Names
Deleted: *Let*
Deleted: Bound Names
Deleted:
Deleted:
Deleted: *BindingList*
Deleted:
Deleted: *BindingList*
Deleted: false
Deleted: <#>RuntimeStatic Semantics String Value¶ <#>*BindingIdentifier* **:** *Identifier* ¶ <#>Return the String Value of *Identifier*
Deleted: Initialization
Deleted: *enviornment*
Deleted: initialization
Deleted: the
Deleted: z
Deleted: z
Deleted: *BindingIdentifier* BindingIdentifier
Deleted: *enviornment*
Deleted:
Deleted: InitializeBinding
Deleted: *enviornment*
Deleted: *environment*
Deleted: *Identifier*
Deleted: Value

a. Let *lhs* be the result of evaluating *Identifier* as described in 11.1.2.
b. Return PutValue(*lhs*, *value*).

Runtime Semantics: Evaluation

LexicalDeclaration **:** *LetOrConst BindingList* **;**

1. Let *next* be the result of evaluating *BindingList*.
2. ReturnIfAbrupt(*next*).
3. Return NormalCompletion(empty).

BindingList **:** *BindingList* **,** *LexicalBinding*

1. Let *next* be the result of evaluating *BindingList*.
2. ReturnIfAbrupt(*next*).
3. Return the result of evaluating *LexicalBinding*.

LexicalBinding **:** *BindingIdentifier*

1. Let *env* be the running execution context's LexicalEnvironment.
2. Return the result of performing Binding Initialisation for *BindingIdentifier* passing **undefined** and *env* as the arguments.

NOTE A static semantics rule ensures that this form of *LexicalBinding* never occurs in a `const` declaration.

LexicalBinding **:** *BindingIdentifier Initialiser*

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *value* be GetValue(*rhs*).
3. ReturnIfAbrupt(*value*).
4. Let *env* be the running execution context's LexicalEnvironment.
5. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.

LexicalBinding **:** *BindingPattern Initialiser*

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *value* be ToObject(GetValue(*rhs*)).
3. ReturnIfAbrupt(*value*).
4. Let *env* be the running execution context's LexicalEnvironment.
5. Return the result of performing Binding Initialisation for *BindingPattern* using *value* and *env* as the arguments.

12.2.2 Variable Statement

NOTE A `var` statement declares variables that are scoped to the running execution context's VariableEnvironment. Var variables are created when their containing Lexical Environment is instantiated and are initialised to **undefined** when created. Within the scope of any VariableEnvironemnt a common *Identifier* may appear in more than one *VariableDeclaration* but those declarations collective define only one variable. A variable defined by a *VariableDeclaration* with an *Initialiser* is assigned the value of its *Initialiser*'s *AssignmentExpression* when the *VariableDeclaration* is executed, not when the variable is created.

Syntax

VariableStatement **:**
 `var` *VariableDeclarationList* **;**

VariableDeclarationList **:**
 VariableDeclaration
 VariableDeclarationList **,** *VariableDeclaration*

Deleted: Call
Deleted: The production
Deleted: *LetDeclaration* … *LetOrConst* `let` *Let*
Deleted: is evaluated as follows:
Deleted: Evaluate *Let*
Deleted: If *next* is an abrupt completion, return *next*
Deleted: (normal, …mpty, empty)
Deleted: *LetBindingList* **:** *LetLexicalBinding* is evaluated as follows:¶
<#>EvaluateReturn the result of evaluating *LetLexicalBinding*.¶
Deleted: *Let*…*indingList* **:** *Let*…*indingList* **,** *Let*
Deleted: is evaluated as follows:
Deleted: Evaluate *Let*
Deleted: If *next* is an abrupt completion, return *next*
Deleted: *Evaluate* Let
Deleted: *Let*
Deleted: is evaluated as follows:
Deleted: running
Deleted: current
Deleted: Perform …inding Initialization
Deleted: e
Deleted: *The production*
Deleted: *Let*
Deleted: is evaluated as follows:
Deleted: If *value* is an abrupt completion, return *rval*.
Deleted: running
Deleted: current
Deleted: Perform …inding Initialization
Deleted: *Let*
Deleted: is evaluated as follows:
Deleted: *rval*
Deleted: If *rval* is an abrupt completion,
Deleted: *rval*
Deleted: running
Deleted: current
Deleted: Perform …inding Initialization
Deleted: Evaluate
Deleted: *rval*
Deleted: as the *obj* parameter …nd *env* as the *environme*
Deleted: <#>*BindingIdentifier* **:** *Identifier* is evaluated as follows:¶
Deleted: <#>The production *Initialiser* : =
Deleted: <#>The *LetBindingListNoIn*,
Deleted: <#>12.2.2 Const Declaration¶
Deleted: current

VariableDeclarationListNoIn **:**
VariableDeclarationNoIn
VariableDeclarationListNoIn **,** *VariableDeclarationNoIn*

VariableDeclaration **:**
BindingIdentifier Initialiser$_{opt}$
BindingPattern Initialiser

VariableDeclarationNoIn **:**
BindingIdentifier InitialiserNoIn$_{opt}$
BindingPattern InitialiserNoIn

The semantics of the *VariableDeclarationListNoIn*, *VariableDeclarationNoIn* and *InitialiserNoIn* productions are the same as the *VariableDeclarationList*, *VariableDeclaration* and *Initialiser* productions except that the contained *VariableDeclarationListNoIn*, *VariableDeclarationNoIn*, *InitialiserNoIn* and *AssignmentExpressionNoIn* are used in of the contained *VariableDeclarationList*, *VariableDeclaration*, *Initialiser* and *AssignmentExpression*, respectively.

12.2.2.1 Static Semantics

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 Initialiser*$_{opt}$

1. Return the BoundNames of *BindingIdentifier*.

VariableDeclaration **:** *BindingPattern Initialiser*

1. Return the BoundNames of *BindingPattern*.

12.2.2.2 Runtime Semantics

Runtime Semantics: Binding Initialisation

With arguments *value* and *environment*.

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

VariableDeclaration **:** *BindingIdentifier*

1. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and **undefined** as the arguments.

VariableDeclaration **:** *BindingIdentifier Initialiser*

1. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and **undefined** as the arguments.

VariableDeclaration **:** *BindingPattern Initialiser*

Deleted: ¶

Deleted: NOTE A `var` statement declares variables that are scoped to th… current execution context's VariableEnvironment. Var variables are created when their containing Lexical Environment is instantiated and are initialised to undefined when created. Within the scope of any VariableEnvironemnt a common *Identifi…* may appear in more than one *VariableDeclaration* but those declaration… collective define only one variable. A variable defined by a *VariableDeclaration* with an *Initialiser* is assigned the value … its *Initialiser's AssignmentExpression* when the *VariableDeclaration* is executed, not when the variable is created. ¶
Semantics¶

Deleted: Static Semantics: Early Errors¶
VariableDeclaration **:** *BindingPattern Initialiser*
It is a Syntax Error if the source code matching this production is not contained i… extended code.¶

Deleted: The BoundNames of the production *VariableStatement* **:** `var` *VariableDeclarationLis…* is determined as follows:¶
Return BoundNames of *VariableDeclarationList*.¶
VariableDeclarationList **:** *VariableDeclaration* ¶
Return BoundNames of *VariableDeclaratio…*

Deleted: the

Deleted: z

Deleted: ize

1. Return the result of performing Binding Initialisation for *BindingPattern* passing *value* and **undefined** as the arguments.

Runtime Semantics: Evaluation

VariableStatement **:** **`var`** *VariableDeclarationList* **;**

1. Let *next* be the result of evaluating *VariableDeclarationList*.
2. ReturnIfAbrupt(*next*).
3. Return NormalCompletion(empty).

VariableDeclarationList **:** *VariableDeclarationList* **,** *VariableDeclaration*

1. Let *next* be the result of evaluating *VariableDeclarationList*.
2. ReturnIfAbrupt(*next*).
3. Return the result of evaluating *VariableDeclaration*.

VariableDeclaration **:** *BindingIdentifier*

1. Return NormalCompletion(empty).

VariableDeclaration **:** *BindingIdentifier Initialiser*

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *value* be GetValue(*rhs*).
3. ReturnIfAbrupt(*value*).
4. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and **undefined** as the arguments.

NOTE If a *VariableDeclaration* is nested within a with statement and the *Identifier* in the *VariableDeclaration* is the same as a property name of the binding object of the with statement's object environment record, then step 3 will assign value to the property instead of to the VariableEnvironment binding of the *Identifier*.

VariableDeclaration **:** *BindingPattern Initialiser*

2. Let *rhs* be the result of evaluating *Initialiser*.
3. Let *rval* be ToObject(GetValue(*rhs*)).
4. ReturnIfAbrupt(*rval*).
5. Return the result of performing Binding Initialisation for *BindingPattern* passing *rval* and **undefined** as arguments.

12.2.3 Destructuring Binding Patterns

Syntax

BindingPattern **:**
  ObjectBindingPattern
  ArrayBindingPattern

ObjectBindingPattern **:**
  { }
  { *BindingPropertyList* **}**
  { *BindingPropertyList* **, }**

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

Deleted: The production
Deleted: is evaluated as follows:
Deleted: Evaluate
Deleted: If *next* is an abrupt completion, return *next*
Deleted: (normal,
Deleted: y, empty)
Deleted: The BoundNames of the production *VariableDeclarationList* **:** *VariableDeclaration* is determine as follows:¶
<#>Return BoundNames of *VariableDeclaration*.¶
Deleted: The production *VariableDeclarationList* **:** *VariableDeclaration* is evaluated as follows:¶
<#>Return the result of evaluating Evaluate *VariableDeclaration*.¶
Deleted: The BoundNames of the production *VariableDeclarationList* **:** *VariableDeclarationList* **,** *VariableDeclaration* is determined as follows:¶
<#>Let *names* be BoundNames of *VariableDeclarationLi*
<#>Append to *names* the elements of BoundNames of *VariableDeclaration*.¶
<#>Return *names*.¶
Deleted: is evaluated as follows:
Deleted: Evaluate
Deleted: If *next* is an abrupt completion, return *next*
Deleted: Evaluate
Deleted: The BoundNames of the production
Deleted: is evaluated as follows:
Deleted: the String value of *BindingIdentifier*
Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *value* is an abrupt completion, return *value*
Deleted: Perform
Deleted: Initialization
Deleted: NOTE 1 The String value of a
Deleted: 2
Deleted: 4
Deleted: The static semantics of the production
Deleted: is evaluated as follows:
Deleted: ToObject(
Deleted:)
Deleted: If *rval* is an abrupt completion, return *rval*
Deleted: Perform
Deleted: Initialization
Deleted: Evaluate
Deleted: using
Deleted: as the *obj* parameter
Deleted: the *environment* parameter
Deleted: <#>The *VariableDeclarationListNoIn*,
Deleted: *VariableStatement* **:**¶

BindingPropertyList **:**
BindingProperty
BindingPropertyList **,** *BindingProperty*

BindingElementList **:**
Elision$_{opt}$ *BindingElement*
BindingElementList **,** *Elision*$_{opt}$ *BindingElement*

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

BindingElement **:**
SingleNameBinding
BindingPattern Initialiser$_{opt}$

SingleNameBinding **:**
BindingIdentifier Initialiser$_{opt}$

BindingRestElement **:**
... *BindingIdentifier*

12.2.3.1 Static Semantics

Static Semantics: Early Errors

BindingPattern **:** *ObjectBindingPattern*

- It is a Syntax Error if the BoundNames of *ObjectBindingPattern* contains the string "`eval`" or the string "`arguments`".

BindingPattern **:** *ArrayBindingPattern*

- It is a Syntax Error if the BoundNames of *ArrayBindingPattern* contains the string "`eval`" or the string "`arguments`".

Static Semantics: BoundNames

ObjectBindingPattern **: { }**

1. Return an empty List.

ArrayBindingPattern **: [** *Elision*$_{opt}$ **]**

1. Return an empty List.

ArrayBindingPattern **: [** *Elision*$_{opt}$ *BindingRestElement* **]**

1. Return the BoundNames of *BindingRestElement*.

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

1. Return the BoundNames of *BindingElementList*.

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

1. Let *names* be BoundNames of *BindingElementList*.
2. Append to *names* the elements of BoundNames of *BindingRestElement*.
3. Return *names*.

Deleted: *Identifier*

Comment [AWB1689]: Note that this may a computed property name

Deleted: *PropertyName* : *SingleNameBind*

Deleted: *PropertyName* : *BindingPattern Initialiser*$_{opt}$

Deleted: *Identifier* .

Deleted: ...

Deleted: *Identifier*

Deleted: Bound Names

Deleted: *Object*

Deleted: Bound Names

Deleted: Bound Names

Deleted: Semantics¶

Deleted: A variable statement declares variables that are created as defined in 10.5. Variables are initialised to **undefined** when created. A variable with an *Initialiser* is assign the value of its *AssignmentExpression* when the *VariableStatement* is executed, not when the variable is created.¶
Semantics¶
The production *VariableStatement* **: var** *VariableDeclarationList* **;** is evaluated as follo
<#>Evaluate *VariableDeclarationList*.¶
<#>Return (normal, empty, empty).¶
The production *VariableDeclarationList* **:** *VariableDeclaration* is evaluated as follows:¶
<#>Evaluate *VariableDeclaration*.¶
The production *VariableDeclarationList* **:** *VariableDeclarationList* **,** *VariableDeclaration* i evaluated as follows:¶
<#>Evaluate *VariableDeclarationList*.¶
<#>Evaluate *VariableDeclaration*.¶
The production *VariableDeclaration* **:** *Identifier* evaluated as follows:¶
<#>Return a String value containing the same sequence of characters as in the *Identifier*.¶
The production *VariableDeclaration* **:** *Identifier Initialiser* is evaluated as follows:¶
<#>Let *lhs* be the result of evaluating *Identifier* described in 11.1.2.¶
<#>Let *rhs* be the result of evaluating *Initialise*
<#>Let *value* be GetValue(*rhs*).¶
<#>Call PutValue(*lhs*, *value*).¶
<#>Return a String value containing the same sequence of characters as in the *Identifier*.¶

Deleted: The BoundNames of the production *BindingPattern* **:** *ObjectBindingPattern* is determined as follows:¶

Comment [AW90]: The destructuring wiki page argues for allowing empty binding patte I'm not really convinced.

Deleted: *ObjectBindingPattern* **:**¶
{ *BindingPropertyList* **}**
{ *BindingPropertyList* **, }**¶

Deleted: *List*

Deleted: *ArrayBindingPattern* **:**
[*BindingElementList* **]** ¶

BindingPropertyList **:** *BindingPropertyList* **,** *BindingProperty*

1. Let *names* be BoundNames of *BindingPropertyList*.
2. Append to *names* the elements of BoundNames of *BindingProperty*.
3. Return *names*.

BindingElementList **:** *Elision*$_{opt}$ *BindingElement*

1. Return BoundNames of *BindingElement*.

BindingElementList **:** *BindingElementList* **,** *Elision*$_{opt}$ *BindingElement*

1. Let *names* be BoundNames of *BindingElementList*.
2. Append to *names* the elements of BoundNames of *BindingElement*.
3. Return *names*.

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

1. Return the BoundNames of *BindingElement*.

SingleNameBinding **:** *BindingIdentifier Initialiser*$_{opt}$

1. Return the BoundNames of *BindingIdentifier*.

BindingElement **:** *BindingPattern Initialiser*$_{opt}$

1. Return the BoundNames of *BindingPattern*.

Static Semantics: HasInitialiser

BindingElement **:** *BindingPattern*

1. Return **false**.

BindingElement **:** *BindingPattern Initialiser*

1. Return **true**.

SingleNameBinding **:** *BindingIdentifier*

1. Return **false**.

SingleNameBinding **:** *BindingIdentifier Initialiser*

1. Return **true**.

12.2.3.2 Runtime Semantics

Runtime Semantics: Binding Initialisation

With parameters *value* and *environment*.

NOTE When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialised in order to deal with the possibility of multiple parameters with the same name.

BindingPattern **:** *ObjectBindingPattern*

1. Assert: Type(*value*) is Object

Deleted: *BindingPropertyList* **:** *BindingProperty* ¶
<#>Return BoundNames of *BindingProperty*.¶

Deleted: *BindingProperty* **:** *SingleNameBinding* ¶
<#>Return the BoundNames of *SingleNameBinding* .¶

Deleted: *BindingElement* **:** *SingleNameBinding* ¶
<#>Return the BoundNames of *SingleNameBinding* .¶

Deleted: *BindingRestElement* **:** ... *BindingIdentifier* ¶
<#>Return the BoundNames of *BindingIdentifier*.¶

Deleted: *BindingElement* **:** *SingleNameBinding* ¶
<#>Return HasInitializer of *SingleNameBinding* .¶

Deleted: Initialization

Deleted: *enviornment*

Deleted: initialization

Deleted: the

Deleted: z

Deleted: The production

Deleted: If *value* is neither of **null** or **undefined**, then

2. Return the result of performing Binding Initialisation for *ObjectBindingPattern* using *value* and *environment* as arguments.

BindingPattern : *ArrayBindingPattern*

1. Assert: Type(*value*) is Object.
2. Return the result of performing Indexed Binding Initialisation for *ArrayBindingPattern* using *value*, 0, and *environment* as arguments.

ObjectBindingPattern : { }

1. Return NormalCompletion(empty).

BindingPropertyList : *BindingPropertyList* , *BindingProperty*

1. Let *status* be the result of performing Binding Initialisation for *BindingPropertyList* using *value* and *environment* as arguments.
2. ReturnIfAbrupt(*status*).
3. Return the result of performing Binding Initialisation 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 Keyed Binding Initialisation 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 Keyed Binding Initialisation for *BindingElement* using *value*, *environment*, and *P* as arguments.

Runtime Semantics: Indexed Binding Initialisation

With parameters *array*, *nextIndex*, and *environment*.

NOTE When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialised in order to deal with the possibility of multiple parameters with the same name.

ArrayBindingPattern : [$Elision_{opt}$]

1. Return NormalCompletion(empty).

ArrayBindingPattern: [$Elision_{opt}$ *BindingRestElement*]

1. Let *nextIndex* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing Indexed Binding Initialisation for *BindingRestElement* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: [*BindingElementList*]

1. Return the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: [*BindingElementList* , $Elision_{opt}$]

Deleted: <#>Let *obj* be ToObject(*value*).¶
Deleted: is evaluated with parameters *valu*
Deleted: Perform …inding Initialization
Deleted: *value* as the
Deleted: *obj*
Deleted: parameter …nd *env* as the …*nvironm*
Deleted: The BoundNames of the production
Deleted: If *value* is neither of **null** or **undefin**
Deleted: <#>Let *array* be ToObject(*value*
Deleted: <#>Let *lenValue* be the result of
Deleted: <#>Else,¶
Deleted: <#>Let *arrayLength* be 0.¶
Deleted: is evaluated with parameters *valu*
Deleted: Perform …ndexed Binding Initializa
Deleted: Evaluate …*rrayBindingPattern* using
Deleted: *array*
Deleted:
Deleted: arrayLength, **undefined**
Deleted: as the *obj* parameter …nd *env* as the
Deleted: The BoundNames of the production
Deleted: with parameters *value* and *env* as
Deleted: Return
Comment [AW91]: The destructuring wiki
Deleted: *ObjectBindingPattern* **:**¶
Deleted: The BoundNames of the production
Deleted: The production *BindingPropertyList*
Deleted: The BoundNames of the production
Deleted: with parameters *obj* and *env* as
Deleted: *next*
Deleted: Perform …inding Initialization
Deleted: Evaluate …*indingPropertyList* using
Deleted: *next*
Deleted: If *next* is an abrupt completion, retur
Deleted: Perform …inding Initialization
Deleted: Evaluate …*indingProperty* using
Deleted: Perform …eyed Binding Initializatio
Deleted: PropName of
Deleted: Perform …eyed Binding Initializatio
Deleted: Evaluate
Deleted: Initialization
Deleted: *arrayLength*,
Deleted: initialization
Deleted: the
Deleted: z
Deleted: Return
Deleted: If *Elision* is present, then let *nextInde*
Deleted: Perform …ndexed Binding Initializa
Deleted: *arrayLength*,
Deleted: Perform …ndexed Binding Initializa

1. Return the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: `[` *BindingElementList* `,` *Elision*$_{opt}$ *BindingRestElement* `]`

1. Let *next* be the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.
2. ReturnIfAbrupt(*next*).
3. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Return the result of performing Indexed Binding Initialisation for *BindingRestElement* using *array*, *next*+*skip* , and *environment* as arguments.

BindingElementList **:** *Elision*$_{opt}$ *BindingElement*

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *status* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *nextIndex*+*skip* , and *environment* as arguments.
3. ReturnIfAbrupt(*status*).
4. Return *nextIndex* +*skip*+1.

BindingElementList **:** *BindingElementList* **,** *Elision*$_{opt}$ *BindingElement*

1. Let *listNext* be the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.
2. ReturnIfAbrupt(*listNext*).
3. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Let *status* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *listNext*+*skip* , and *environment* as arguments.
5. ReturnIfAbrupt(*status*).
6. Return *listNext* +*skip*+1.

BindingElement **:** *SingleNameBinding*

1. Return the result of performing Keyed Binding Initialisation for *SingleNameBinding* using *array*, *environment*, and ToString(*nextIndex*) as the arguments.

BindingElement **:** *BindingPattern Initialiser*$_{opt}$

1. Let *P* be ToString(*nextIndex*).
2. Let *v* be the result of Get(*array*, *P*).
3. ReturnIfAbrupt(*v*).
4. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be ToObject(GetValue(*defaultValue*)).
5. ReturnIfAbrupt(*v*).
6. Return the result of performing Binding Initialisation for *BindingPattern* passing *v* and *environment* as arguments.

BindingRestElement **:** `...` *BindingIdentifier*

2. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
3. Let *lenVal* be the result of Get(*array*, `"length"`).
4. Let *arrayLength* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*arrayLength*).
6. Let *n*=0.
7. Let *index* = *nextIndex*.
8. Repeat, while *index* < *arrayLength*
   a. Let *P* be ToString(*index*).
   b. Let *exists* be the result of HasProperty(*array*, *P*).

Deleted: Perform …ndexed Binding Initialization
Deleted: 0,
Deleted: *index*
Deleted: Initialization
Deleted: ,
Deleted: 0
Deleted: *index*
Deleted: If *index* is an abrupt completion, return *index*.
Deleted: If *Elision* is present, then let *skip* be the result o
Deleted: Perform …ndexed Binding Initialization
Deleted: *arrayLength*,
Deleted: *index*
Deleted: The BoundNames of the production
Deleted: with the parameters *obj*, *env*, and *index* as
Deleted: If *Elision* is present, then let *skip* be the result o
Deleted: *next*
Deleted: Perform …ndexed Binding Initialization
Deleted: Evaluate …*indingElement* using *obj*…*rray* as th
Deleted: *arrayLength*,
Deleted:
Deleted: *env* as the *env* parameter, and *index*+*skip* …s th
Deleted: *next*
Deleted: If *next* is an abrupt completion, return *next*.
Deleted: *index*
Deleted: The BoundNames of the production
Deleted: is evaluated with the parameters *obj*, *env*, a
Deleted: Initialization
Deleted: *obj*
Deleted: *arrayLength*,
Deleted: as the *obj* parameter, *env* as the *env* parameter,
Deleted: *next*
Deleted: If *next* is an abrupt completion, return *next*
Deleted: If *Elision* is present, then let *skip* be the result o
Deleted: Evaluate
Deleted: *next*
Deleted: Perform …ndexed Binding Initialization
Deleted: *obj*…*rray* as the *obj* parameter
Deleted: *arrayLength*,
Deleted: *env* as the *env* parameter
Deleted: ,
Deleted: and …*istNext*+*skip* , and environment as the ind
Deleted: *listNext*
Deleted: If *listNext* is an abrupt completion, return *listNe*
Deleted: *listNext*
Deleted: ¶
Deleted: The BoundNames of the productions
Deleted: <#>If *nextIndex* ≥ *arrayLength*, then¶
Deleted: with the parameters *obj*, *env* and *index* as
Deleted: Perform …eyed Binding Initialization
Deleted: Evaluate
Deleted: *P*, *obj*…*rray*, *environment*, and ToString(*env*
Deleted: The BoundNames of the production
Deleted: <#>If *nextIndex* ≥ *arrayLength*, then¶
Deleted: <#>Let *exists* be the result of calling the
Deleted: with the parameters *obj*, *env* and *index* as

c. ReturnIfAbrupt(*exists*).

d. If *exists* is **true**, then

i. Let *v* be the result of Get(*array*, *P*).

ii. ReturnIfAbrupt(*v*).

iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

e. Let *n* = *n*+1.

f. Let *index* = *index*+1.

9. Return the result of performing Binding Initialisation for *BindingIdentifier* using *A* and *environment* as arguments.

Runtime Semantics: Keyed Binding Initialisation

With parameters *obj*, *environment*, and *propertyName*.

NOTE When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialised in order to deal with the possibility of multiple parameters with the same name.

BindingElement **:** *BindingPattern Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be ToObject(GetValue(*defaultValue*)).
4. ReturnIfAbrupt(*v*).
5. Return the result of performing Binding Initialisation for *BindingPattern* passing *v* and *environment* as arguments.

SingleNameBinding **:** *BindingIdentifier Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be GetValue(*defaultValue*).
4. ReturnIfAbrupt(*v*).
5. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *v* and *environment* as arguments.

12.3 Empty Statement

Syntax

EmptyStatement **:**
  ;

12.3.1.1 Runtime Semantics

Runtime Semantics: Evaluation

EmptyStatement **:** **;**

1. Return NormalCompletion(empty).

Deleted: calling the [[…et]]… internal meth

Deleted: If *v* is an abrupt completion, ret *v*

Deleted: ,

Deleted: .[[value]]

Deleted: , and **false**

Deleted: Perform …inding Initialization

Deleted: *array*

Deleted: Initialization

Deleted: initialization

Deleted: the

Deleted: z

Deleted: *BindingElement*: *SingleNameBinding* <#>Return the result of performing Perform Keyed Binding InitializationInitialisation for *SingleNameBinding* using *obj*, *environment*, an *propertyName* as the arguments.¶

Deleted: <#>Let exists be the result of calling the [[HasProperty]] internal method of obj with argument propertyName. ¶
<#>If exists is true, then¶
<#>Let v be the result of calling the [[Get]] internal method of obj with argument propertyName.¶

Deleted: calling the [[…et]]… internal metho

Deleted: <#>Else ¶

Deleted: *v*

Deleted: <#>Else,¶
<#> Let v be **undefined**.¶

Deleted: If *v* is an abrupt completion, return *v*.

Deleted: Perform …inding Initialization

Deleted: <#>Let *exists* be the result of calling the [[HasProperty]] internal method of *obj* with argument *propertyName*.¶
<#>If *exists* is **true**, then¶
<#>Let *v* be the result of calling the [[Get]] internal method of *obj* passing *propertyName* as argument.¶

Deleted: calling the [[…et]] internal method o

Deleted: <#>Else ¶

Deleted: *v*

Deleted: <#>Else,¶
<#> Let *v* be **undefined**.¶

Deleted: If *v* is an abrupt completion, return *v*.

Deleted: Perform …inding

Deleted: <#>The BoundNames of the production *BindingRestElement* : . . . *BindingIdentifier* is determined as follows

Deleted: <#>The BoundNames of the

Deleted: <#>The production

Deleted: Semantics¶

Deleted: (normal, …mpty, empty).

12.4 Expression Statement

Syntax

ExpressionStatement **:**
    [lookahead ∉ {**{**, **function**, **class** }] *Expression* **;**

NOTE An *ExpressionStatement* cannot start with an opening curly brace because that might make it ambiguous with a *Block*. Also, an *ExpressionStatement* cannot start with the **function** or **class** keywords because that would make it ambiguous with a *FunctionDeclaration*, a *GeneratorDeclaration*, or a *ClassDeclaration*.

12.4.1.1 Runtime Semantics

Runtime Semantics: Evaluation

ExpressionStatement **:** [lookahead ∉ {**{**, **function**, **class** }] *Expression* **;**

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *value* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*value*).
4. Return NormalCompletion(*value*).

12.5 The `if` Statement

Syntax

IfStatement **:**
    if (*Expression* **)** *Statement* **else** *Statement*
    if (*Expression* **)** *Statement*

Each **else** for which the choice of associated **if** is ambiguous shall be associated with the nearest possible **if** that would otherwise have no corresponding **else**.

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*.

12.5.1.1 Runtime Semantics

Runtime Semantics: Evaluation

IfStatement **: if (** *Expression* **)** *Statement* **else** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
3. ReturnIfAbrupt(*exprValue*).
4. If *exprValue* is **true**, then
    a. Let *stmtValue* be the result of evaluating the first *Statement*.
5. Else,
    a. Let *stmtValue* be the result of evaluating the second *Statement*.
6. If *stmtValue*.[[type]] is normal and *stmtValue*.[[value]] is empty, then
    a. Return NormalCompletion(**undefined**).

Deleted: might

Deleted: Semantics¶ The production

Deleted: is evaluated as follows:

Deleted: If *value* is an abrupt completion, return *value*

Deleted: (normal, *GetValue(exprRef)*

Deleted: , empty

Deleted: Semantics¶ The VarDeclaredNames of the production

Deleted: is determined as follows:

Deleted: The production

Deleted: is evaluated as follows:

Deleted: If *exprValue* is an abrupt completion, return *exprValue*

Deleted: ToBoolean(GetValue(*exprRef*))

Deleted: Return

Deleted: Return

Comment [AWB1392]: Breaking change from ES5: completion reform

7. Return *stmtValue*.

IfStatement : **if** (*Expression*) *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
3. ReturnIfAbrupt(*exprValue*).
4. If *exprValue* is **false**, then
   a. Return NormalCompletion(**undefined**).
5. Else,
   a. Let *stmtValue* be the result of evaluating *Statement*.
6. If *stmtValue*.[[type]] is normal and *stmtValue*.[[value]] is empty, then
   a. Return NormalCompletion(**undefined**).
7. Return *stmtValue*.

12.6 Iteration Statements

Syntax

IterationStatement :
- **do** *Statement* **while** (*Expression*) ; opt
- **while** (*Expression*) *Statement*
- **for** (*ExpressionNoIn*opt ; *Expression*opt ; *Expression*opt) *Statement*
- **for** (**var** *VariableDeclarationListNoIn* ; *Expression*opt ; *Expression*opt) *Statement*
- **for** (*LexicalDeclarationNoIn* ; *Expression*opt ; *Expression*opt) *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*

ForDeclaration :
- *LetOrConst ForBinding*

NOTE 1 *ForBinding* is defined in 11.1.4.2.

NOTE 2 A semicolon is not required after a **do-while** statement.

Runtime Semantics

Runtime Semantics: LoopContinues Abstract Operation

The abstract operation LoopContinues with arguments *completion* and *labelSet* is defined by the following step:

1. If *completion*.[[type]] is normal, then return **true**.
2. If *completion*.[[type]] is not continue, then return **false**.
3. If *completion*.[[target]] is empty, then return **true**.
4. If *completion*.[[target]] is an element of *labelSet*, then return **true**.
5. Return **false**.

NOTE Within the *Statement* part of an *IterationStatement* a *ContinueStatement* may be used to begin a new iteration.

12.6.1 The do-while Statement

Static Semantics: VarDeclaredNames

IterationStatement : **do** *Statement* **while** (*Expression*) ; opt

Deleted: The VarDeclaredNames of the production *IfStatement* : **if** (*Expression*) *Statement* is determined as follows:¶ <#>Return the VarDeclaredNames of *Statement*. The production

Deleted: is evaluated as follows:

Deleted: If *exprValue* is an abrupt completion return *exprValue*

Deleted: ToBoolean(GetValue(*exprRef*))

Comment [AWB693]: Breaking change from ES5: completion reform

Deleted: return

Deleted: (normal, **empty**

Deleted: , empty).

Comment [AWB1394]: Breaking change from ES5: completion reform

Deleted: <#>Return the result of evaluating *Statement*.¶

Comment [AWB695]: This is breaking change from ES5 which allowed a *VariableDeclarationNoIn* to appear here. See discuss thread "lexical for-in/for-of loose end"

Deleted: *BindingIdentifer*

Deleted: *VariableDeclarationNoIn*

Deleted: *ForVarDeclaration*

Deleted: *Expression*

Deleted: *ForVarDeclaration* :¶ *BindingPattern* *VariableDeclarationNoIn*¶

Comment [AWB696]: Note that this is technically a breaking change from ES5, however it is made to match web reality.

Deleted: ForBinding :¶ BindingIdentifier BindingPattern ¶

Formatted: Note

Deleted: Within the *Statement* part of an *IterationStatement* a *ContinueStatement* may be used be begin a new iterations. This semantic is defined using t

Deleted: The

Deleted: loopContinues

Deleted: be

Deleted: s

Deleted: The VarDeclaredNames of the production

Deleted: is determined as follows:

1. Return the VarDeclaredNames of *Statement*.

12.6.1.1 **Runtime Semantics**

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement **:** **do** *Statement* **while (** *Expression* **) ;** $_{opt}$

1. Let *V* = **undefined**.
2. Repeat
   a. Let *stmt* be the result of evaluating *Statement*.
   b. If *stmt*.[[value]] is not empty, let *V* = *stmt*.[[value]].
   c. If *stmt* is an abrupt completion and LoopContinues (*stmt*,*labelSet*) is **false**, return *stmt*.
   d. Let *exprRef* be the result of evaluating *Expression*.
   e. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
   f. If *exprValue* is **false**, Return NormalCompletion(*V*).
   g. Else if *exprValue* is not **true**, then
      i. Assert: *exprValue* is an abrupt completion.
      ii. If LoopContinues (*exprValue*,*labelSet*) is **false**, return *exprValue*.

12.6.2 The `while` Statement

Static Semantics: VarDeclaredNames

IterationStatement **: while (** *Expression* **)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

12.6.2.1 **Runtime Semantics**

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement **: while (** *Expression* **)** *Statement*

1. Let *V* = **undefined**.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
   c. If *exprValue* is **false**, return NormalCompletion(*V*).
   d. If *exprValue* is not **true**, then
      i. Assert: *exprValue* is an abrupt completion.
      ii. If LoopContinues (*exprValue*,*labelSet*) is **false**, return *exprValue*.
   e. Let *stmt* be the result of evaluating *Statement*.
   f. If *stmt*.[[value]] is not empty, let *V* = *stmt*.[[value]].
   g. If LoopContinues (*stmt*,*labelSet*) is **false**, return *stmt*.

12.6.3 The `for` Statement

12.6.3.1 **Static Semantics**

Static Semantics: VarDeclaredNames

IterationStatement **: for (** *ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

Deleted: Runtime Semantics: Evaluation¶

Deleted: The production

Deleted:

Deleted: is evaluated as follows:

Comment [AWB697]: Breaking change: completion reform

Deleted: empty

Deleted: <#>Let *iterating* be **true**.¶

Deleted: ,

Deleted: while *iterating* is **true**

Deleted: <#>If *stmt*.type is not continue or|| *stmt*.tar is not in the current label set, then¶
<#>If *stmt*.type is break and *stmt*.target is in the current label set, return (normal, *V*, empty).¶

Deleted: loopContinues

Deleted: ToBoolean(GetValue(*exprRef*))

Deleted: set *iterating* to **false**

Comment [AWB698]: Break/continue/return in the expression works normally (future for do {} or block lam expressions)

Deleted: a Completion Record

Deleted: loopContinues

Deleted: <#>Return (normal, *V*, empty);¶

Deleted: The VarDeclaredNames of the production

Deleted: *Statement* is determined as follows:

Deleted: Runtime Semantics: Evaluation¶

Deleted: The production

Deleted: is evaluated as follows:

Comment [AWB699]: ES5 breaking change: completion reform

Deleted: empty

Deleted: ToBoolean(GetValue(*exprRef*))

Deleted: (normal,

Deleted: , empty

Comment [AWB6100]: Break/continue/return in the expression works normally (future for do {} or block lam expressions)

Deleted: Else i

Deleted: a Completion Record

Deleted: loopContinues

Deleted: <#>If *stmt*.type is not continue or|| *stmt*.tar is not in the current label set, then¶
<#>If *stmt*.type is break and *stmt*.target is in the current label set, then¶
<#>Return (normal, *V*, empty).¶

Deleted: *stmt* is an abrupt completion

Deleted: loopContinues

Comment [AWB 3101]: The lexical scoping of for

Deleted: The VarDeclaredNames of the production

Deleted: is determined as follows:

IterationStatement **:** **for (var** *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Let *names* be BoundNames of *VariableDeclarationListNoIn*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*.

IterationStatement **:** **for (***LexicalDeclarationNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

12.6.3.2 Runtime Semantics

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement **:** **for (***ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*

1. If *ExpressionNoIn* is present, then
   a. Let *exprRef* be the result of evaluating *ExpressionNoIn*.
   b. Let *exprValue* be GetValue(*exprRef*).
   c. If LoopContinues(*exprValue*,*labelSet*) is **false**, return *exprValue*.
2. Return the result of performing For Body Evaluation with the first *Expression* as the *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.

IterationStatement **:** **for (var** *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Let *varDcl* be the result of evaluating *VariableDeclarationListNoIn*.
2. If LoopContinues(*varDcl*,*labelSet*) is **false**, return *varDcl*.
3. Return the result of performing For Body Evaluation with the first *Expression* as the *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.

IterationStatement **:** **for (** *LexicalDeclarationNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *loopEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
3. Let *isConst* be the result of performing IsConstantDeclaration of *LexicalDeclarationNoIn*.
4. For each element *dn* of the BoundNames of *LexicalDeclarationNoIn* do
   a. If *isConst* is **true**, then
      i. Call *loopEnv*'s CreateImmutableBinding concrete method passing *dn* as the argument.
   b. Else,
      i. Call *loopEnv*'s CreateMutableBinding concrete method passing *dn* and **false** as the arguments.
5. Set the running execution context's LexicalEnvironment to *loopEnv*.
6. Let *forDcl* be the result of evaluating *LexicalDeclarationNoIn*.
7. If LoopContinues(*forDcl*, *labelSet*) is **false**, then
   a. Set the running execution context's LexicalEnvironment to *oldEnv*.
   b. Return *forDcl*.
8. Let *bodyResult* be the result of performing For Body Evaluation with the first *Expression* as the *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.
9. Set the running execution context's LexicalEnvironment to *oldEnv*.
10. Return *bodyResult*.

Runtime Semantics: For Body Evaluation Abstract Operation

The abstract operation For Body Evaluation with arguments *testExpr*, *incrementExpr*, *stmt*, and *labelSet* is performed as follows:

Deleted: Runtime Semantics: Evaluation¶
Deleted: The production
Deleted: .
is evaluated as follows:
Deleted: .
Deleted: Call
Deleted: (This value is not used but the call may have side-effects.)
Deleted: loopContinues
Deleted: *increment*
Deleted: *testExpr*
Comment [AWB6102]: ES5 breaking change: Completion reform
Deleted: The abstraction operation For Body Evaluation with arguments *testExpr*, *incrementExpr*, and *labelSet* is performed as follows:¶
<#>Let *V* = **undefined**,empty.¶
<#>Repeat¶
<#>If the first *Expression testExpr* is not [empty]present, then¶
<#>Let *testExprRef* be the result of evaluating the first *testExprExpression*.¶
<#>Let *testExprValue* be ToBoolean(GetValue(*testExprRef*))¶
<#>If *testExprValue*ToBoolean(GetValue(*testEx ef*)) is **false**, return NormalCompletion((normal, *V*, empty).¶
<#>Else if loopContinuesLoopContinues (*testExprValue*,*labelSet*) is **false**, return *testExprValue*.¶
Deleted: loopContinues
Deleted: *incrementExpr*
Deleted: *testExpr*
Comment [AWB6103]: A final decision ha
Deleted: <#>Evaluate
Deleted: running
Deleted: current
Deleted: the
Deleted: *d*
Deleted: running
Deleted: current
Deleted: loopContinues
Deleted: *varDcl*
Deleted: running
Deleted: current
Deleted: *varDcl*
Deleted: *increment*
Deleted: *testExpr*
Deleted: running
Deleted: current

1. Let *V* = **undefined**.
2. Repeat
   a. If *testExpr* is not [empty], then
      i. Let *testExprRef* be the result of evaluating *testExpr*.
      ii. Let *testExprValue* be ToBoolean(GetValue(*testExprRef*))
      iii. If *testExprValue* is **false**, return NormalCompletion(*V*).
      iv. Else if LoopContinues (*testExprValue*,*labelSet*) is **false**, return *testExprValue*.
   b. Let *result* be the result of evaluating *stmt*.
   c. If *result*.[[value]] is not empty, let *V* = *result*.[[value]].
   d. If LoopContinues (*result*,*labelSet*) is **false**, return *result*.
   e. If *incrementExpr* is not [empty], then
      i. Let *incExprRef* be the result of evaluating *incrementExpr*.
      ii. Let *incExprValue* be GetValue(*incExprRef*).
      iii. If LoopContinues(*incExprValue*,*labelSet*) is **false**, return *incExprValue*.

12.6.4 The `for-in` and `for-of` Statements

12.6.4.1 Static Semantics

Static Semantics: Early Errors

IterationStatement **:**
  `for (`*LeftHandSideExpression* `in` *Expression* `)` *Statement*
  `for (`*LeftHandSideExpression* `of` *AssignmentExpression* `)` *Statement*

- It is a Syntax Error *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 the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.
- It is a Syntax Error if the *LeftHandSideExpression* is *CoverParenthesisedExpressionAndArrowParameterList* **:** `(` *Expression* `)` and *Expression* derives a production that would produce a Syntax Error according to these rules if that production is substituted for *LeftHandSideExpression*. This rule is recursively applied.

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

IterationStatement **:**
  `for (`*ForDeclaration* `in` *Expression* `)` *Statement*
  `for (`*ForDeclaration* `of` *AssignmentExpression* `)` *Statement*

- It is a Syntax Error if any element of the BoundNames of *ForDeclaration* also occurs in the VarDeclaredNames of *Statement*.

Static Semantics: BoundNames

ForDeclaration **:** *LetOrConst ForBinding*

1. Return the BoundNames of *ForBinding*.

Static Semantics: VarDeclaredNames

Comment [AWB6104]: ES5 breaking change: Completion reform

Deleted: ,

Deleted: <#>It is a Syntax Error if the *IterationStatement* is contained in strict code and *LeftHandSideExpression* is the *Identifier* `eval` or the *Identifier* `arguments`.¶
<#>It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that statically resolves to a declarative environment record binding and the resolved binding is an immutable binding.¶

Deleted: z

Deleted: d

Deleted: <#>It is a Syntax Error if the *LeftHandSideExpression* is *PrimaryExpression* : `(` *Expression* `)` and *Expression* derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.¶
<#>It is a Syntax Error if IsInvalidAssignmentPattern of *LeftHandSideExpression* is **true**.It is a Syntax Error if the *LeftHandSideExpression* is an *ObjectLiteral* or an *ArrayLiteral* and the source code corresponding to *LeftHandSideExpression* cannot be parsed using *AssignmentPattern* as the goal symbol.¶
I

Deleted: Lexically Declared Names

Deleted: Var Declared Names

Deleted: Bound Names

Deleted: *ForVarDeclaration* **:** *BindingPattern* ¶
<#>Return the Bound Names of *BindingPattern*.¶
ForVarDeclaration **:** *VariableDeclarationNoIn* ¶
<#>Return the Bound Names of *VariableDeclarationNoIn*

Deleted: Bound Names

Deleted: *ForBinding* **:** *BindingIdentifier* ¶
<#>Return the Bound Names of *BindingIdentifier*.¶
ForDeclaration **:** *BindingPattern*¶
<#>Return the Bound Names of *BindingPattern*.¶

Deleted: Var Declared Names

IterationStatement : **for (*LeftHandSideExpression* in *Expression*)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement : **for (var** *ForBinding* **in** *Expression* **)** *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*

IterationStatement : **for (** *ForDeclaration* **in** *Expression* **)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement : **for (** *LeftHandSideExpression* **of** *AssignmentExpression* **)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement : **for (var** *ForBinding* **of** *AssignmentExpression* **)** *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*

IterationStatement : **for (** *ForDeclaration* **of** *AssignmentExpression* **)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

12.6.4.2 Runtime Semantics

Runtime Semantics: Binding Instantiation

With arguments *value* and *environment*.

ForDeclaration : *LetOrConst* *ForBinding*

1. For each element *name* of the BoundNames of *ForBinding* do
   a. If IsConstantDeclaration of *LetOrConst* is **false**, then
      i. Call *environment*'s CreateMutableBinding concrete method with argument *name*.
   b. Else,
      i. Call *environment*'s CreateImmutableBinding concrete method with argument *name*.
2. Return the result of performing Binding Initialisation for *ForBinding* passing *value* and *environment* as the arguments.

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement : **for (** *LeftHandSideExpression* **in** *Expression* **)** *Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Expression*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *LeftHandSideExpression*, *Statement*, *keyResult*, assignment, and *labelSet*.

IterationStatement : **for (var** *ForBinding* **in** *Expression* **)** *Statement*

Deleted: The VarDeclaredNames of the production
Deleted: is determined as follows:
Deleted: Var Declared Names
Deleted: *VariableDeclarationNoIn*
Deleted: *BindingIdentifier*
Deleted: Bound Names
Deleted: *Variable*
Deleted: *BindingIdentifier*
Deleted: *DeclarationNoIn*
Deleted: Var Declared Names
Deleted: Var Declared Names
Deleted: Var Declared Names
Deleted: *ForVarDeclaration*
Deleted: Bound Names
Deleted: *ForVarDeclaration*
Deleted: Var Declared Names
Deleted: Var Declared Names
Deleted: Bound Names
Deleted: Perform
Deleted: **Runtime Semantics: Binding Initialisation¶**
With arguments *value* and *environment*.¶
NOTE **undefined** is passed for *environment* to indicate that a PutValue operation should be used to assign the initialisation value. This is the the case for **var** statements formal parameter lists of non-strict functions. In thos cases a lexical binding is hosted and preinitialized prior to evaluation of its initialize
ForVarDeclaration : *BindingPattern* ¶
<#>Perform Binding Initialisation for *BindingPattern* passing *value* and *env* as the arguments.¶
ForVarDeclaration : *VariableDeclarationNoIn* ¶
<#>Perform Binding Initialisation for *VariableDeclarationNoIn* passing *value* and *env* the arguments.¶
ForBinding : *BindingIdentifier* ¶
<#>Perform Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.¶
ForBinding: *BindingPattern*¶
<#>Perform Binding Initialisation for *BindingPattern* passing *value* and *env* as the arguments.¶
Deleted: **Runtime Semantics: Evaluation¶**
Deleted: The production
Deleted: is evaluated as follows:
Deleted: *Statement*
Deleted: .[[value]]
Deleted: *BindingIdentifier*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Expression*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForBinding*, *Statement*, *keyResult*, varBinding, and *labelSet*.

IterationStatement **:** `for` `(`*ForDeclaration* `in` *Expression* `)` *Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Expression*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForDeclaration*, *Statement*, *keyResult*, lexicalBinding, and *labelSet*.

IterationStatement **:** `for` `(` *LeftHandSideExpression* `of` *AssignmentExpression* `)` *Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *AssignmentExpression*, iterate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *LeftHandSideExpression*, *Statement*, *keyResult*, assignment, and *labelSet*.

IterationStatement **:** `for` `(` `var` *ForBinding* `of` *AssignmentExpression* `)` *Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *AssignmentExpression*, iterate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForBinding*, *Statement*, *keyResult*, varBinding, and *labelSet*.

IterationStatement **:** `for` `(`*ForDeclaration* `of` *AssignmentExpression* `)` *Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *AssignmentExpression*, iterate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForDeclaration*, *Statement*, *keyResult*, lexicalBinding, and *labelSet*.

Runtime Semantics: For In/Of Expression Evaluation Abstract Operation

The abstract operation For In/Of Expression Evaluation is called with arguments *expr*, *iterationKind*, and *labelSet*. The value of *iterationKind* is either **enumerate** or **iterate**.

1. Let *exprRef* be the result of evaluating the production that is *expr*.
2. Let *exprValue* be GetValue(*exprRef*).
3. If *exprValue* is an abrupt completion,
   a. If LoopContinues(*exprValue*,*labelSet*) is **false**, then return *exprValue*.
   b. Else, return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.
4. If *exprValue*.[[value]] is **null** or **undefined**, return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.
5. Let *obj* be ToObject(*exprValue*).
6. If *iterationKind* is enumerate, then
   a. Let *keys* be the result of calling the [[Enumerate]] internal method of *obj* with no arguments.
7. Else,
   a. Assert *iterationKind* is iterate.
   b. Let *iterator* be the result of performing Invoke with arguments *obj*, @@iterator and an empty List.
   c. Let *keys* be ToObject(*iterator*).
8. If *keys* is an abrupt completion, then
   a. If LoopContinues(*exprValue*,*labelSet*) is **false**, then return *exprValue*.

Deleted: *Statement*
Deleted: *BindingIdentifier*
Deleted: .[[value]]
Deleted: *Statement*
Deleted: .[[value]]
Deleted: ForDeclaration
Deleted: *Statement*
Deleted: .[[value]]
Deleted: *BindingIdentifier*
Deleted: *Statement*
Deleted: *BindingIdentifier*
Deleted: .[[value]]
Deleted: *Statement*
Deleted: .[[value]]
Deleted: ForDeclaration
Deleted: ion
Deleted: is performed as follows:
Comment [AWB15105]: NOTE that if this is a for-le this places the evaluation of the AssignmentExpressio outside the scope of the bindings. Perhaps in should b
Deleted: *Expression*
Deleted: *e*
Deleted: *e*
Deleted: then return
Deleted: loopContinues
Deleted: *e*
Deleted: *e*
Comment [AWB6106]: Note a continue in the initiali
Comment [AWB6107]: Completion value reform
Deleted: **undefined**,
Comment [AWB6108]: Completion value reform
Deleted: *e*
Deleted: **undefined**,
Deleted:
Deleted: .
Deleted: (normal, empty, empty)
Deleted: *e*
Deleted: arguments **true** and **true**
Deleted: *keys*
Deleted: %
Deleted: %,
Comment [AWB6109]: Break/continue in the
Deleted: <#>Let *keys* be the result of calling the
Deleted: LoopContinues
Deleted: *e*
Deleted: *e*

b. Assert: *keys*.[[type]] is continue
c. Return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.

9. Return *keys*.

Runtime Semantics: For In/Of Body Evaluation Abstract Operation

The abstract operation For In/Of Body Evaluation is called with arguments *lhs*, *stmt*, *keys*, *lhsKind*, and *labelSet*. The value of *lhsKind* is either **assignment**, **varBinding** or **lexicalBinding**.

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *noArgs* be an empty List.
3. Let *V* = **undefined**.
4. Repeat
   a. Let *nextResult* be the result of Invoke(*keys*, "`next`").
   b. ReturnIfAbrupt(*nextResult*).
   c. If Type(*nextResult*) is not Object, then throw a **TypeError** exception.
   d. Let *done* be IteratorComplete(*nextResult*).
   e. ReturnIfAbrupt(*done*).
   f. If *done* is **true**, then return NormalCompletion(*V*).
   g. Let *nextValue* be the result of IteratorValue(*nextResult*).
   h. ReturnIfAbrupt(*nextValue*).
   i. If *lhsKind* is assignment, then
      i. Assert: *lhs* is a *LeftHandSideExpression*.
      ii. If *lhs* is neither an *ObjectLiteral* nor an *ArrayLiteral*, then
         1. Let *lhsRef* be the result of evaluating *lhs* (it may be evaluated repeatedly).
         2. Let *status* be the result of performing PutValue(*lhsRef*, *nextValue*).
      iii. Else
         1. Let *AssignmentPattern* be the parse of the source code corresponding to *lhs* using *AssignmentPattern* as the goal symbol.
         2. Let *rval* be ToObject(*nextValue*).
         3. If *rval* is an abrupt completion, then let *status* be *rval*.
         4. Else, let *status* be the result of performing Destructuring Assignment Evaluation of *AssignmentPattern* using *rval* as the argument.
   j. Else if *lhsKind* is varBinding, then
      i. Assert: *lhs* is a *ForBinding*.
      ii. Let *status* be the result of performing Binding Initialisation for *lhs* passing *nextValue* and **undefined** as the arguments.
   k. Else,
      i. Assert *lhsKind* is lexicalBinding.
      ii. Assert: *lhs* is a *ForDeclaration*.
      iii. Let *iterationEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
      iv. Perform Binding Instantiation for *lhs* passing *nextValue* and *iterationEnv* as arguments.
      v. Let *status* be NormalCompletion(empty)
      vi. Set the running execution context's LexicalEnvironment to *iterationEnv*.
   l. If *status*.[[type]] is normal, then
      i. Let *status* be the result of evaluating *stmt*.
      ii. If *status*.[[type]] is normal and *status*.[[value]] is not empty, then
         1. Let *V* = *status*.[[value]].
   m. Set the running execution context's LexicalEnvironment to *oldEnv*.
   n. If *status* is an abrupt completion and LoopContinues(*status*,*labelSet*) is **false**, then return *status*.

12.7 The `continue` Statement

Syntax

ContinueStatement **:**
   `continue ;`
   `continue` [no *LineTerminator* here] *Identifier* **;**

Comment [AWB6110]: Note a continue in initializer expression is just like a break

Comment [AWB6111]: Completion value reform

Deleted: undefined,

Deleted: ion

Deleted: is performed as follows:

Deleted: running

Deleted: current

Comment [AWB6112]: Completion value reform

Deleted:)

Deleted: empty

Deleted: *next*

Deleted: performing

Deleted: with arguments

Deleted: , *keys*, and an empty arguments Li

Deleted: *inner*

Deleted: IteratorComplete(*next*)

Deleted: <#>If loopContinuesLoopContinues(*next*,*labelSet*) **false**, then return *next*.¶
<#>If *next* is an abrupt completion, then let *status* be *next*.¶
<#>Else,¶
<#>Assert *next*.[[type]] is normal.¶
<#>Let *nextValue* be *next*.[[value]].¶

Deleted: <#>Let *P* be the name of th next property of *obj* whose [[Enumerab attribute is **true**. If there is no such property, return (normal, *V*, empty).¶

Deleted: the *LeftHandSideExpression*

Deleted: Call

Deleted: *P*

Comment [AWB15113]: Recent

Comment [AWB6114]: ToDo, update this

Deleted: ForDeclaration

Comment [AWB11115]: Need to understa

Deleted: running

Deleted: current

Deleted: *stmt*

Deleted: *Statement*

Deleted: If *stmt*.value is not empty, l

Deleted: *stmt*

Deleted: running

Deleted: current

Deleted: loopContinues

Deleted: <#>If *stmt*.type is break an

Deleted: <#>The VarDeclaredNames o

Deleted: <#>The production

12.7.1.1 Static Semantics

Static Semantics: Early Errors

ContinueStatement **:** `continue ;`

- It is a Syntax Error if this production is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.

ContinueStatement **:** `continue` [no *LineTerminator* here] *Identifier* `;`

- It is a Syntax Error if *Identifier* does not appear in the *CurrentLabelSet* of an enclosing (but not crossing function boundaries) *IterationStatement*.

Deleted: Semantics¶

Deleted: A program is considered syntactically incorre if either of the following is true:¶

Deleted: The program contains a **`continue`** statement without the optional *Identifier*, which

Deleted: The program contains a **`continue`** statement with the optional *Identifier*, where

Deleted: label set

12.7.1.2 Runtime Semantics

Runtime Semantics: Evaluation

ContinueStatement **:** `continue ;`

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

ContinueStatement **:** `continue` [no *LineTerminator* here] *Identifier* `;`

1. Return Completion {[[type]]: continue, [[value]]: empty, [[target]]: *Identifier*}.

Deleted: A *ContinueStatement* without an *Identifier* is evaluated as follows:¶

Deleted: (

Deleted:).

Deleted: A *ContinueStatement* with the optional *Identifie* evaluated as follows:¶

Deleted: (

Deleted:).

12.8 The `break` Statement

Syntax

BreakStatement **:**
 `break ;`
 `break` [no *LineTerminator* here] *Identifier* `;`

12.8.1.1 Static Semantics

Static Semantics: Early Errors

BreakStatement **:** `break ;`

- It is a Syntax Error if this production is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement* or a *SwitchStatement*.

BreakStatement **:** `break` [no *LineTerminator* here] *Identifier* `;`

- It is a Syntax Error if *Identifier* does not appear in the *CurrentLabelSet* of an enclosing (but not crossing function boundaries) *Statement*.

Deleted: Semantics¶
A program is considered syntactically incorrect if either the following is true:¶

Deleted: The program contains a **`break`** statemer without the optional *Identifier*, which is

Deleted: The program contains a **`break`** statemer with the optional *Identifier*, where

Deleted: label set

12.8.1.2 Runtime Semantics

Runtime Semantics: Evaluation

BreakStatement **:** `break ;`

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

BreakStatement **:** `break` [no *LineTerminator* here] *Identifier* `;`

1. Return Completion {[[type]]: break, [[value]]: empty, [[target]]: *Identifier*}.

Deleted: A *BreakStatement* without an *Identifier* is evaluated as follows:¶

Deleted: (

Deleted:).

Deleted: A *BreakStatement* with an *Identifier* is evaluate as follows:¶

Deleted: (

Deleted:).

12.9 The `return` Statement

Syntax

ReturnStatement **:**
    `return ;`
    `return` [no *LineTerminator* here] *Expression* **`;`**

NOTE A **`return`** statement causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is **undefined**. Otherwise, the return value is the value of *Expression*.

Static Semantics

Static Semantics: Early Errors

- It is a Syntax Error if a **`return`** statement is not within a *FunctionBody* or a *GeneratorBody*.

12.9.1.1 Runtime Semantics

Runtime Semantics: Evaluation

ReturnStatement **:** **`return ;`**

1. Return Completion {[[type]]: return, [[value]]: **undefined**, [[target]]: empty}.

ReturnStatement **:** **`return`** [no *LineTerminator* here] *Expression* **;**

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*exprValue*).
4. Return Completion {[[type]]: return, [[value]]: *exprValue*, [[target]]: empty}.

12.10 The `with` Statement

Syntax

WithStatement **:**
    `with (` *Expression* **`)`** *Statement*

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.

Static Semantics

Static Semantics: Early Errors

WithStatement **:** **`with (`** *Expression* **`)`** *Statement*

- It is a Syntax Error if the code that matches this production is contained in strict code.

Static Semantics: VarDeclaredNames

WithStatement **:** **`with (`** *Expression* **`)`** *Statement*

1. Return the VarDeclaredNames of *Statement*.

Deleted: <#>Semantics¶

Deleted: An ECMAScript program is considered syntactically incorrect if it conta

Deleted: that

Deleted: 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*.

Deleted: A *ReturnStatement* is evaluated as follows:¶
If the *Expression* is not present, r

Deleted: (

Deleted:).

Comment [AWB9116]: TODO If exprRef is Reference that invokes a getter we probably should find a way to specify that the get call i handled as a tail call.

Deleted: If *exprValue* is an abrupt completion return *exprValue*

Deleted: (

Deleted: GetValue(

Deleted: *exprRef*)

Deleted:).

Deleted: current

Deleted: Semantics¶
The static semantics of the production

Deleted: are:

Deleted: static

Deleted: The VarDeclaredNames of the production

Deleted: is determined as follows:

12.10.1.1 Runtime Semantics

Runtime Semantics: Evaluation

WithStatement **:** `with` **(** *Expression* **)** *Statement*

1. Let *val* be the result of evaluating *Expression*.
2. Let *obj* be ToObject(GetValue(*val*)).
3. ReturnIfAbrupt(*obj*).
4. Let *oldEnv* be the running execution context's LexicalEnvironment.
5. Let *newEnv* be the result of calling NewObjectEnvironment passing *obj* and *oldEnv* as the arguments.
6. Set the *withEnvironment* flag of *newEnv* to **true**.
7. Set the running execution context's LexicalEnvironment to *newEnv*.
8. Let *C* be the result of evaluating *Statement*.
9. Set the running execution context's Lexical Environment to *oldEnv*.
10. Return *C*.

NOTE No matter how control leaves the embedded *Statement*, whether normally or by some form of abrupt completion or exception, the LexicalEnvironment is always restored to its former state.

12.11 The `switch` Statement

Syntax

SwitchStatement **:**
  `switch` **(** *Expression* **)** *CaseBlock*

CaseBlock **:**
  { $CaseClauses_{opt}$ **}**
  { $CaseClauses_{opt}$ *DefaultClause* $CaseClauses_{opt}$ **}**

CaseClauses **:**
  CaseClause
  CaseClauses CaseClause

CaseClause **:**
  `case` *Expression* **:** $StatementList_{opt}$

DefaultClause **:**
  `default` **:** $StatementList_{opt}$

12.11.1.1 Static Semantics

Static Semantics: Early Errors

CaseBlock **: { *CaseClauses* }**

- It is a Syntax Error if the LexicallyDeclaredNames of *CaseClauses* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *CaseClauses* also occurs in the VarDeclaredNames of *CaseClauses*.

Static Semantics: LexicalDeclarations

CaseBlock **: { }**

1. Return a new empty List.

CaseBlock **: {** $CaseClauses_{opt}$ *DefaultClause* $CaseClauses_{opt}$ **}**

Deleted: The production
Deleted: is evaluated as follows:
Deleted: If *obj* is an abrupt completion, return *ob*
Deleted: *j*
Deleted: running
Deleted: current
Deleted: *provideThis*
Deleted: running
Deleted: current
Deleted: but if an exception is thrown during the evaluat let *C* be (throw, *V*, empty), where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
Deleted: running
Deleted: current
Deleted: <#>12.10.1 Strict Mode Restrictions¶ <#>Strict mode code may not include a *WithStatement*. The occurrence of a *WithStatement* i such a context is treated as a SyntaxError.¶
Deleted: are:
Deleted: i
Deleted: *CaseClause* **:** `case` *Expression* **:** *StatementList* <#>It is a Syntax Error if *StatementList* includes a *StatementListItem* **:** *FunctionDeclaration* production ar the source code matching this *CaseClause* productior not contained in extended code.¶
Deleted: *CaseBlock* **: {** *CaseClauses* **}** ¶ <#>Return the LexicalDeclarations of *CaseClauses*.¶

1. If the first *CaseClauses* is present, let *declarations* be the LexicalDeclarations of the first *CaseClauses*.
2. Else let *declarations* be a new empty List.
3. Append to *declarations* the elements of the LexicalDeclarations of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *declarations*.
5. Else return the result of appending to *declarations* the elements of the LexicalDeclarations of the second *CaseClauses*.

CaseClauses **:** *CaseClauses CaseClause*

1. Let *declarations* be LexicalDeclarations of *CaseClauses*.
2. Append to *declarations* the elements of the LexicalDeclarations of *CaseClause*.
3. Return *declarations*.

CaseClause **:** **`case`** *Expression* **:** *StatementList*$_{opt}$

1. If the *StatementList* is present, return the LexicalDeclarations of *StatementList*.
2. Else return a new empty List.

DefaultClause **:** **`default`** **:** *StatementList*$_{opt}$

1. If the *StatementList* is present, return the LexicalDeclarations of *StatementList*.
2. Else return a new empty List.

Static Semantics: LexicallyDeclaredNames

CaseBlock **: { }**

1. Return a new empty List.

CaseBlock **: {** *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

1. If the first *CaseClauses* is present, let *names* be the LexicallyDeclaredNames of the first *CaseClauses*.
2. Else let *names* be a new empty List.
3. Append to *names* the elements of the LexicallyDeclaredNames of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Else return the result of appending to *names* the elements of the LexicallyDeclaredNames of the second *CaseClauses*.

CaseClauses **:** *CaseClauses CaseClause*

1. Let *names* be LexicallyDeclaredNames of *CaseClauses*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *CaseClause*.
3. Return *names*.

CaseClause **:** **`case`** *Expression* **:** *StatementList*$_{opt}$

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

DefaultClause **:** **`default`** **:** *StatementList*$_{opt}$

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

Static Semantics: VarDeclaredNames

SwitchStatement **: `switch` (** *Expression* **)** *CaseBlock*

1. Return the VarDeclaredNames of *CaseBlock*.

Deleted: LexicallyDeclaredNames
Deleted: *names*
Deleted: *names*
Deleted: the
Deleted: lyDeclaredNames
Deleted: *CaseClauses* : *CaseClause* ¶ <#>Return the LexicalDeclarations of *CaseClause*.¶
Deleted: *names*
Deleted: *CaseBlock* : { *CaseClauses* } ¶ <#>Return the LexicallyDeclaredNames of *CaseClauses*.¶
Deleted: the
Deleted: *CaseClauses* : *CaseClause* ¶ <#>Return the LexicallyDeclaredNames of *CaseClause*.¶
Deleted: Semantics¶ The VarDeclaredNames of the production
Deleted: is determined as follows:

CaseBlock **: { }**

1. Return a new empty List.

CaseBlock **: {** *CaseClauses*$_{opt}$ *DefaultClause* *CaseClauses*$_{opt}$ **}**

1. If the first *CaseClauses* is present, let *names* be the VarDeclaredNames of the first *CaseClauses*.
2. Else let *names* be a new empty List.
3. Append to *names* the elements of the VarDeclaredNames of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Else return the result of appending to *names* the elements of the VarDeclaredNames of the second *CaseClauses*.

CaseClauses **:** *CaseClauses CaseClause*

1. Let *names* be VarDeclaredNames of *CaseClauses*.
2. Append to *names* the elements of the VarDeclaredNames of *CaseClause*.
3. Return *names*.

CaseClause **: `case`** *Expression* **:** *StatementList*$_{opt}$

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Else return a new empty List.

DefaultClause **: `default` :** *StatementList*$_{opt}$

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Else return a new empty List.

12.11.1.2 Runtime Semantics

Runtime Semantics: Case Block Evaluation

With argument *input*.

CaseBlock **: {** *CaseClauses*$_{opt}$ **}**

1. Let *V* = **undefined**.
2. Let *A* be the list of *CaseClause* items in source text order.
3. Let *searching* be **true**.
4. Repeat, while *searching* is **true**
   a. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, return NormalCompletion(*V*).
   b. Let *clauseSelector* be the result of evaluating *C*.
   c. ReturnIfAbrupt(*clauseSelector*).
   d. Let *matched* be the result of performing Strict Equality Comparison *input* === *clauseSelector*.
   e. If *matched* is **true**, then
      i. Set *searching* to **false**.
      ii. If *C* has a *StatementList*, then
         1. Evaluate *C*'s *StatementList* and let *R* be the result.
         2. ReturnIfAbrupt(*R*).
         3. Let *V* = *R*.[[value]].
5. Repeat
   a. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, return NormalCompletion(*V*).
   b. If *C* has a *StatementList*, then
      i. Evaluate *C*'s *StatementList* and let *R* be the result.
      ii. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
      iii. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.

Deleted: *CaseBlock* : { *CaseClauses* } ¶
<#>Return the VarDeclaredNames of *CaseClauses*.¶

Deleted: the

Deleted: *CaseClauses* : *CaseClause* ¶
<#>Return the VarDeclaredNames of *CaseClause*.¶

Deleted: Runtime Semantics: Evaluation¶
The production *SwitchStatement* **: `switch` (** *Expression* **)** *CaseBlock* is evaluated as follows:¶
<#>Let *exprRef* be the result of evaluating *Expression*.¶
<#>Let *oldEnv* be the running execution context's LexicalEnvironment.¶
<#>Let *blockEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argum
<#>Perform Block Declaration Instantiation using *CaseB* and *blockEnv*.¶
<#>Let *R* be the result of evaluating *CaseBlock*, passing i GetValue(*exprRef*) as a parameter.¶
<#>Set the running execution context's LexicalEnvironme to *oldEnv*.¶
<#>If *R*.type is break and *R*.target is in the current label s return (normal, *R*.value, empty).¶
<#>Return *R*.¶
NOTE No matter how control leaves the *SwitchStateme* the LexicalEnvironment is always restored to its former state.¶

Deleted: The production

Deleted: is given an input parameter, *input*, and is evaluated as follows:

Comment [AWB6117]: ES5 breaking change: completion reform

Deleted: empty

Deleted: (normal,

Deleted: , empty

Deleted: If *clauseSelector* is an abrupt completion, ret *clauseSelector*

Deleted: *input*

Deleted: equal to *clauseSelector* as defined by the Stri Equality Comparision Algorithm (11.9.61)

Deleted: === operator

Deleted: *r*

Deleted: If *R* is an abrupt completion, then retur

Deleted: (normal,

Deleted: , empty

Deleted: (

Deleted:).

CaseBlock : { *CaseClauses*$_{opt}$ *DefaultClause* *CaseClauses*$_{opt}$ }

1. Let *V* = **undefined**.
2. Let *A* be the list of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Let *found* be **false**.
4. Repeat letting *C* be in order each *CaseClause* in *A*
   a. If *found* is **false**, then
      i. Let *clauseSelector* be the result of Case Selector Evaluation of *C*.
      ii. If *clauseSelector* is an abrupt completion, then
         1. If *clauseSelector*.[[value]] is empty, then return Completion {[[type]]: *clauseSelector*.[[type]], [[value]]: **undefined**, [[target]]: *clauseSelector*.[[target]]}.
         2. Else, return *clauseSelector*.
      iii. Let *found* be the result of performing Strict Equality Comparison *input* === *clauseSelector*.
   b. If *found* is **true**, then
      i. Evaluate *CaseClause C* and let *R* be the result.
      ii. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
      iii. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
5. Let *foundInB* be **false**.
6. If *found* is **false**, then
   a. Let *B* be a new List containing the *CaseClause* items in the second *CaseClauses*, in source text order.
   b. Repeat, letting *C* be in order each *CaseClause* in *B*
      i. If *foundInB* is **false**, then
         1. Let *clauseSelector* be the result of Case Selector Evaluation of *C*.
         2. If *clauseSelector* is an abrupt completion, then
            a If *clauseSelector*.[[value]] is empty, then return Completion {[[type]]: *clauseSelector*.[[type]], [[value]]: **undefined**, [[target]]: *clauseSelector*.[[target]]}.
            b Else, return *clauseSelector*.
         3. Let *foundInB* be the result of performing Strict Equality Comparison *input* === *clauseSelector*.
      ii. If *foundInB* is **true**, then
         1. Evaluate *CaseClause C* and let *R* be the result.
         2. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
         3. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
7. If *foundInB* is **true**, then return NormalCompletion(*V*).
8. Evaluate *DefaultClause* and let *R* be the result.
9. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
10. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
11. Let *B* be a new List containing the *CaseClause* items in the second *CaseClauses*, in source text order.
12. Repeat, letting *C* be in order each *CaseClause* in *B* (NOTE this is another complete iteration of the second *CaseClauses*)
   b. Evaluate *CaseClause C* and let *R* be the result.
   c. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
   d. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
13. Return NormalCompletion(*V*).

Runtime Semantics: Case Selector Evaluation

CaseClause : **case** *Expression* : *StatementList*$_{opt}$

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return GetValue(*exprRef*).

NOTE Case Selector Evaluation 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.

Deleted: The production
Deleted: is given an input parameter, *input*, and is evaluated as follows:
Comment [AWB6118]: ES5 breaking change: completion reform
Deleted: .empty.
Deleted: <#>Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.¶
Deleted: evaluating
Formatted
Deleted: <#>If *input* is equal to *clauseSelector* as defined by the Strict Equality ComparisonComparison Algorith (11.9.61)=== operator, then set *found* to tr
Deleted: <#>If *C* has a *StatementList*, t
Deleted: 's *StatementList*
Deleted: (
Deleted:).
Deleted: 1
Deleted: of
Deleted: while *foundInB* is **false** and all elements of *B* have not been processed
Deleted: <#>Let *C* be the next *CaseClause* in *B*. ¶
Deleted: evaluating
Deleted: .
Formatted
Deleted: <#>If *input* is equal to *clauseSelector* as defined by the Strict Equality ComparisonComparision Algorithm (11.9.61) === operator, then <#>Sset *foundInB* to **true**.¶
Deleted: If *C* has a *StatementList*,
Deleted: *C*'s *StatementList*
Deleted: (
Deleted:).
Deleted: <#>If *foundInB* is **false** and the
Deleted: the
Deleted: 's *StatementList*
Deleted: (*R*.type, *V*, *R*.target)
Deleted: 1
Deleted: of
Deleted: Note
Deleted: that if step 7.a.i has been performed t
Deleted: <#>Let *C* be the next *CaseClaus*
Deleted: 's *StatementList*
Deleted: return (*R*.type, *V*, *R*.target)
Deleted: Evaluating
Deleted: *CaseClause*

Runtime Semantics: Evaluation

SwitchStatement **: `switch` `(`** *Expression* **`)`** *CaseBlock*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *switchValue* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*switchValue*).
4. Let *oldEnv* be the running execution context's LexicalEnvironment.
5. Let *blockEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
6. Perform Block Declaration Instantiation using *CaseBlock* and *blockEnv*.
7. Let *R* be the result of performing Case Block Evaluation of *CaseBlock* with argument *switchValue*.
8. Set the running execution context's LexicalEnvironment to *oldEnv*.
9. Return *R*.

NOTE No matter how control leaves the *SwitchStatement* the LexicalEnvironment is always restored to its former state.

CaseClause **: `case`** *Expression* **:**

1. Return NormalCompletion(empty).

CaseClause **: `case`** *Expression* **:** *StatementList*

1. Return the result of evaluating *StatementList*.

DefaultClause **: `default:`**

1. Return NormalCompletion(empty).

DefaultClause **: `default:`** *StatementList*

1. Return the result of evaluating *StatementList*.

12.12 Labelled Statements

Syntax

LabelledStatement **:**
  Identifier **:** *Statement*

NOTE A *Statement* may be prefixed by a label. Labelled statements are only used in conjunction with labelled **`break`** and **`continue`** statements. ECMAScript has no **`goto`** statement. A *Statement* can be part of a *LabelledStatement*, which itself can be part of a *LabelledStatement*, and so on. The labels introduced this way are collectively referred to as the "current label set" when describing the semantics of individual statements. A *LabelledStatement* has no semantic meaning other than the introduction of a label to a *label set*. The label set of an *IterationStatement* or a *SwitchStatement* initially contains the single element **empty**. The label set of any other statement is initially empty.

12.12.1.1 Static Semantics

Static Semantics: Early Errors

- It is a Syntax Error if a *LabelledStatement* is enclosed by a *LabelledStatement* with the same *Identifier* as the enclosed *LabelledStatement*. This does not apply to a *LabelledStatement* appearing within the body of a *FunctionDeclaration* and a *LabelledStatement* that encloses, directly or indirectly, the *FunctionDeclaration*.

Static Semantics: VarDeclaredNames

Deleted: If *switchValue* is an abrupt completion, return *switchValue*

Deleted: running

Deleted: current

Deleted: running

Deleted: current

Deleted: <#>If *R*.[[type]] is break and *R*.[[target]] is in *CurrentLabelSet*, return NormalCompletion(*R*.[[value]]).¶

Deleted: [empty]

Deleted: The production

Deleted: opt

Deleted: is evaluated as follows:

Deleted: [empty]

Deleted: <#>Let *exprRef* be the result of evaluating *Expression*.¶
<#>Return GetValue(*exprRef*).¶
<#>NOTE Evaluating *CaseClause* does not execu[te] the associated *StatementList*. It simply evaluates t[he] *Expression* and returns the value, which the *CaseBlock* algorithm uses to determine which *StatementList* to start executing.¶

Deleted: Semantics¶

Deleted: An ECMAScript program is considered syntactically incorrect if it contains

Deleted: that

Deleted: label

Deleted: labels

Deleted: that

Deleted: is

Deleted: nested

Deleted: ,

Deleted: within

Deleted: a labelled statement

LabelledStatement : *Identifier* : *Statement*

1. Return the VarDeclaredNames of *Statement*.

Deleted: The VarDeclaredNames of the production *DefaultCase*

Deleted: *t* is determined as follows:

12.12.1.2 Runtime Semantics

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

LabelledStatement : *Identifier* : *Statement*

1. Let *label* be the StringValue of *Identifier*.
2. Let *newLabelSet* be a new List containing *label* and the elements of *labelSet*.
3. If *Statement* is either *LabelledStatement* or *BreakableStatement*, then
   a. Let *stmtResult* be the result of performing Labelled Evaluation of *Statement* with argument *newLabelSet*.
4. Else,
   a. Let *stmtResult* be the result of evaluating *Statement*.
5. If *stmtResult*.[[type]] is **break** and *stmtResult*.[[target]] is the same value as *label*, then
   a. Let *result* be NormalCompletion(*stmtResult*.[[value]]).
6. Else,
   a. Let *result* be *stmtResult*.
7. Return *result*.

Deleted:

Deleted: 1

Runtime Semantics: Evaluation

LabelledStatement : *Identifier* : *Statement*

3. Let *newLabelSet* be a new empty List.
4. Return the result of performing Labelled Evaluation of this *LabelledStatement* with argument *newLabelSet*.

Deleted: The production *Identifier* : *Statement* is evaluated by adding *Identifier* to the label set of *Statement* and then evaluating *Statement*. If the *LabelledStatement* itself has a non-empty label set, these labels are also added to the label set of *Statement* before evaluating it. If the result of evaluating *Statement* is (**break**, *V*, *L*) where *L* is equal to *Identifier*, the production results in (**normal**, *V*, **empty**).¶

12.13 The `throw` Statement

Deleted: <#>**Prior to the evaluation of a *LabelledStatement*, the contained *Statement* is regarded as possessing an empty label set, unless it is an *IterationStatement* or a *SwitchStatement*, in which case it is regarded as possessing a label set consisting of the single element, empty.**

Syntax

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

Runtime Semantics: Evaluation

Deleted: **Semantics¶**

The production *ThrowStatement* **:** **`throw`** [no *LineTerminator* here] *Expression* **`;`** is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*exprValue*).
4. Return Completion {[[type]]: throw, [[value]]: GetValue(*exprRef*), [[target]]: empty}.

12.14 The `try` Statement

Syntax

TryStatement **:**
    `try` *Block Catch*
    `try` *Block Finally*
    `try` *Block Catch Finally*

Catch **:**
    `catch (` *CatchParameter* **`)`** *Block*

Deleted: *Identifier*

Finally **:**
  `finally` *Block*

CatchParameter **:**
  BindingIdentifier
  BindingPattern

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.

12.14.1.1 Static Semantics

Static Semantics: Early Errors

Catch **: `catch (`** *CatchParameter* **`)`** *Block*

- It is a Syntax Error if any element of the BoundNames of *CatchParameter* also occurs in the LexicallyDeclaredNames of *Block*.
- It is a Syntax Error if any element of the BoundNames of *CatchParameter* also occurs in the VarDeclaredNames of *Block*.

Static Semantics: VarDeclaredNames

TryStatement **: `try`** *Block Catch*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Catch*.
3. Return *names*.

TryStatement **: `try`** *Block Finally*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Finally*.
3. Return *names*.

TryStatement **: `try`** *Block Catch Finally*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Catch*.
3. Append to *names* the elements of the VarDeclaredNames of *Finally*.
4. Return *names*.

Catch **: `catch (`** *CatchParameter* **`)`** *Block*

1. Return the VarDeclaredNames of *Block*.

12.14.1.2 Runtime Semantics

Runtime Semantics: Binding Initialisation

With arguments *value* and *environment*.

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

CatchParameter **:** *BindingPattern*

Formatted: Note

Deleted: *Identifier*

Deleted: the code that matches this production is contained in extended code and

Comment [AWB119]: Note that this is a new restrict that does not exist in ES5

Deleted: Static Semantics: BoundNames¶
CatchParameter **:** *BindingIdentifier* ¶
<#>Return the BoundNames of *BindingIdentifier*.¶
CatchParameter **:** *BindingPattern* ¶
<#>Return the BoundNames of *BindingPattern*.¶

Deleted: Semantics¶
The VarDeclaredNames of the production

Deleted: is determined as follows:

Deleted: <#>*Finally* : `finally` *Block* ¶
<#>Return the VarDeclaredNames of *Block*.¶

Deleted: Initialization

Deleted: *enviornment*

Deleted: initialization

Deleted: the

Deleted: z

Deleted: z

Deleted: *CatchParameter* **:** *BindingIdentifier*¶
<#>PerformReturn the result of performing Binding InitializationInitialisation for *BindingIdentifier* passing *val* and *enviornmentenvironment* as the arguments.¶

1. Let *exceptionObj* be ToObject(*value*).
2. ReturnIfAbrupt(*exceptionObj*).
3. Return the result of performing Binding Initialisation for *BindingPattern* passing *exceptionObj* and *environment* as the arguments.

Runtime Semantics: Catch Clause Evaluation

with parameter *thrownValue*

Catch **:** **`catch (`** *CatchParameter* **`)`** *Block*

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *catchEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
3. For each element *argName* of the BoundNames of *CatchParameter*, do
   a. Call the CreateMutableBinding concrete method of *catchEnv* passing *argName* as the argument.
4. Let *status* be the result of performing Binding Initialisation for *CatchParameter* passing *thrownValue* and *catchEnv* as arguments.
5. ReturnIfAbrupt(*status*).
6. Set the running execution context's LexicalEnvironment to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.
8. Set the running execution context's LexicalEnvironment to *oldEnv*.
9. Return *B*.

NOTE No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

Runtime Semantics: Evaluation

TryStatement **:** **`try`** *Block Catch*

1. Let *B* be the result of evaluating *Block*.
2. If *B*.[[type]] is not throw, return *B*.
3. Return the result of performing Catch Clause Evaluation of *Catch* with parameter *B*.[[value]].

TryStatement **:** **`try`** *Block Finally*

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.[[type]] is normal, return *B*.
4. Return *F*.

TryStatement **:** **`try`** *Block Catch Finally*

1. Let *B* be the result of evaluating *Block*.
2. If *B*.[[type]] is throw, then
   a. Let *C* be the result of performing Catch Clause Evaluation of *Catch* with parameter *B*.value.
3. Else *B*.[[type]] is not throw,
   a. Let *C* be *B*.
4. Let *F* be the result of evaluating *Finally*.
5. If *F*.[[type]] is normal, return *C*.
6. Return *F*.

12.15 The `debugger` statement

Syntax

DebuggerStatement **:**
    `debugger ;`

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after 0.25" + Indent at: 0.25"

Comment [AWB10120]: Catching a throw null or undefined with a destructuring paramet rethrows a TypeError. Does this make sense

Deleted: Perform

Deleted: Initialization

Deleted: *enviornment*

Deleted: running

Deleted: current

Deleted: Perform

Deleted: Initialization

Deleted: *g*

Deleted: If *an exception was thrown in the preceding step, return* (throw, *V*, empty) *where is the exception. (Execution now proceeds as if exception were thrown.)status* is an abrupt completion, then return *status*

Deleted: running

Deleted: current

Deleted: running

Deleted: current

Deleted: The production

Deleted: is evaluated as follows:

Deleted: evaluating

Deleted: The production

Deleted: is evaluated as follows:

Deleted: The production

Deleted: is evaluated as follows:

Deleted: evaluating

Deleted: ,

Deleted: <#>The production *Catch* : `catch (` *IdentifierCatchParameter* `)` *Block* evaluated with parameter *C* as follows:¶
<#>Let *C* be the parameter that has bee passed to this production.¶
<#>Let *oldEnv* be the running executio context's LexicalEnvironment.¶
<#>Let *catchEnv* be the result of callin NewDeclarativeEnvironment passing *oldEnv* as the argument. ¶
<#>For each element *argName* of the BoundNames of *CatchParameter*, do¶
<#>Call the CreateMutableBinding concrete method of *catchEnv* passing *argName* the *Identifier* String value as th argument.¶
<#>Call theEvaluate *CatchParameter* SetMutableBinding using concrete meth of *catchEnv* passing the *Identifier*, *C* and *catEnv*, and false as arguments. Note th

Deleted: <#>The production *Finally* :

Deleted: <#>The BoundNames of the

Runtime Semantics: Evaluation

NOTE Evaluating the *DebuggerStatement* production may allow an implementation to cause a breakpoint when run under a debugger. If a debugger is not present or active this statement has no observable effect.

The production *DebuggerStatement* **:** `debugger ;` is evaluated as follows:

1. If an implementation defined debugging facility is available and enabled, then
    a. Perform an implementation defined debugging action.
    b. Let *result* be an implementation defined Completion value.
2. Else
    a. Let *result* be NormalCompletion(empty).
3. Return *result*.

13 Functions and Generators

13.1 Function Definitions

Syntax

FunctionDeclaration **:**
  `function` *BindingIdentifier* `(` *FormalParameters* `)` `{` *FunctionBody* `}`

FunctionExpression **:**
  `function` *BindingIdentifier*$_{opt}$ `(` *FormalParameters* `)` `{` *FunctionBody* `}`

StrictFormalParameters **:**
  FormalParameters

FormalParameters **:**
  [empty]
  FormalParameterList

FormalParameterList **:**
  FunctionRestParameter
  FormalsList
  FormalsList , *FunctionRestParameter*

FormalsList **:**
  FormalParameter
  FormalsList , *FormalParameter*

FunctionRestParameter **:**
  `...` *BindingIdentifier*

FormalParameter **:**
  BindingElement

FunctionBody **:**
  FunctionStatementList

FunctionStatementList **:**
  StatementList$_{opt}$

Supplemental Syntax

The following productions are used as an aid in specifying the semantics of certain ECMAScript language features. They are not used when parsing ECMAScript source code.

Deleted: Semantics¶
Deleted: Runtime Semantics: Evaluation¶
Deleted: (normal,
Deleted: , empty
Deleted: *List*
Deleted: opt
Deleted: *List*
Deleted: opt
Deleted: [empty]
Deleted: . . . *Identifier*
Deleted: . . . *Identifier*
Deleted: *FormalParameterList*
Deleted: *Identifier*
Deleted: *FormalParameterList*
Deleted: *Identifier*
Deleted: *BindingIdentifier Initialiser*$_{opt}$
Deleted: *Pattern*
Deleted: *Initialiser*$_{opt}$
Deleted: opt
Deleted: *SourceElements*$_{opt}$

FunctionBody **:**
ThrowTypeError

ThrowTypeError **:**
[empty]

13.1.1.1 Static Semantics

Static Semantics: Early Errors

FunctionDeclaration **: `function`** *BindingIdentifier* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**
and
FunctionExpression **: `function`** *BindingIdentifier*$_{opt}$ **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

- It is a Syntax Error if *FunctionBody* Contains *YieldExpression* is **true**.
- If the source code matching this production is strict code, the Early Error rules for *StrictFormalParameters* **:** *FormalParameters* are applied.
  It is a Syntax Error if IsSimpleParameterList of *FormalParameters* is **false** and any element of the BoundNames of *FormalParameters* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if any element of the BoundNames of *FormalParameters* also occurs in the LexicallyDeclaredNames of *FunctionBody*.

NOTE The LexicallyDeclaredNames of a *FunctionBody* does not include identifiers bound using var or function declarations. Simple parameter lists bind identifiers as VarDeclaredNames. Parameter lists that contain destructuring patterns, default value initialisers, or a rest parameter bind identifiers as LexicallyDeclaredNames.

StrictFormalParameters **:** *FormalParameters*

- It is a Syntax Error if BoundNames of *FormalParameterList* contains any duplicate elements.
- It is a Syntax Error if BoundNames of *FormalParameterList* contains either **`"eval"`** or **`"arguments"`**.

FormalParameters **:** *FormalParameterList*

- It is a Syntax Error if *FormalParameters* Contains *YieldExpression* is **true**.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains either **`"eval"`** or **`"arguments"`**.
- It is a Syntax Error if the source code matching this production is strict code and BoundNames of *FormalParameterList* contains any duplicate elements.

NOTE Multiple occurrences of the same *Identifier* in a *FormalParamterList* is only allowed for non-strict functions and generator functions that have simple parameter lists.

FunctionStatementList **:** *StatementList*

- It is a Syntax Error if the LexicallyDeclaredNames of *StatementList* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *StatementList* also occurs in the VarDeclaredNames of *StatementList*.

FormalParameter **:** *BindingElement*

- It is a Syntax Error if *BindingElement* Contains *YieldExpression*.

Static Semantics: BoundNames

FunctionDeclaration **: `function`** *BindingIdentifier* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. Return the BoundNames of *BindingIdentifier*.

Deleted: <#>NOTE The identifier of a rest parameter is specified as an *Identi*[fier] rather than a *BindingIdentifier* because i[t] may be the the identifier **`arguments`**.¶

Deleted: Semantics¶

Deleted: *List*

Deleted: *List*

Deleted: It is a Syntax Error if the source code matching this production is extended code and the BoundNames of *BindingIdentifier* also occurs in the VarDeclaredNames of *FunctionBody*.¶
It is a Syntax Error if the source code matching this production is extended code and the BoundNames of *BindingIdentifier* a[lso] occurs in the LexicallyDeclaredNames of *FunctionBody*.¶

Deleted: It is a Syntax Error if the source code matching this production is extended code and any element of the LexicallyDeclaredNames of *FormalParameterList* also occurs in the VarDeclaredNames of *FunctionBody*.¶

Deleted: *List*

Deleted: **true**

Deleted: the source code matching this production is extended code and

Deleted: *List*

Deleted: Lexically

Deleted: <#>It is a Syntax Error if IsSimpleParameterList of *FormalParameter*[s] is **false** and BoundNames of *FormalsParameterList* contains any duplica[te] elements.¶
<#>It is a Syntax Error if IsSimpleParameterList of *FormalParameter*[s] is **false** and BoundNames of *FormalParameterList* contains either **`"eval`**[`"`] or **`"arguments"`**.¶
<#>It is a Syntax Error if the source code matching this production is strict code and BoundNames of *FormalsParameterList* conta[ins] any duplicate elements.¶

Deleted: *List*

Deleted: <#>It is a Syntax Error if *FunctionBody* Contains *YieldExpression*.¶

Deleted: Multiple occurrences of the same *Identifier* in a *FormalParamterList* is only allow[ed]

Deleted: *Body*

Deleted: the source code matching this production is extended code and

Deleted: BoundNames

Deleted: i

Deleted: the source code matching this production is extended code and

Deleted: *FormalParameterList* **:** *FormalsList*¶

Deleted: <#>It is a Syntax Error if the source code parsed with this production is

Deleted: *List*

FormalParameters **:** [empty]

1. Return an empty List.

FormalParameterList **:** *FormalsList* **,** *FunctionRestParameter*

1. Let *names* be BoundNames of *FormalsList*.
2. Append to *names* the BoundNames of *FunctionRestParameter*.
3. Return *names*.

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. Let *names* be BoundNames of *FormalsList*.
2. Append to *names* the elements of BoundNames of *FormalParameter*.
3. Return *names*.

Static Semantics: Contains

With parameter *symbol*.

FunctionDeclaration **:** `function` *BindingIdentifier* `(` *FormalParameters* `)` `{` *FunctionBody* `}`

1. Return **false**.

FunctionExpression **:** `function` *BindingIdentifier*$_{opt}$ `(` *FormalParameters* `)` `{` *FunctionBody* `}`

1. Return **false**.

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

Static Semantics: ExpectedArgumentCount

FormalParameters **:** [empty]

1. Return 0.

FormalParameterList **:** *FunctionRestParameter*

1. Return 0.

FormalParameterList **:** *FormalsList* **,** *FunctionRestParameter*

1. Return the ExpectedArgumentCount of *FormalsList*.

NOTE The ExpectedArgumentCount of a *FormalParameterList* is the number of *FormalParameters* to the left of either the rest parameter or the first *FormalParameter* with an Initialiser. A *FormalParameter* without an initialiser is allowed after the first parameter with an initialiser but such parameters are considered to be optional with **undefined** as their default value.

FormalsList **:** *FormalParameter*

1. If HasInitialiser of *FormalParameter* is **false** return 0
2. Return 1.

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. Let *count* be the ExpectedArgumentCount of *FormalsList*.
2. If HasInitialiser of *FormalsList* is **true** or HasInitialiser of *FormalParameter* is **true**, then return *count*.
3. Return *count*+1.

Static Semantics: HasInitialiser

Deleted: *List*

Deleted: *FormalParameterList* **:** *FunctionRestParameter*¶
<#>Return the BoundNames of *FunctionRestParameter*.¶
FormalParameterList **:** *FormalsList*¶
<#>Return the BoundNames of *FormalsList*.¶

Deleted: *FormalParameterList* **:** **. . .** *BindingIdentifier*¶
<#>Return the BoundNames String Value of *BindingIdentifier*.¶
FormalParameter **:** *BindingElement* ¶
<#>Return the BoundNames of *BindingElement* .¶

Deleted: *List*

Deleted: *List*

Deleted: ¶
[empty]

Deleted: ¶

Deleted: *FormalsList* ,

Deleted: ize

Deleted: are

Deleted: ize

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. If HasInitialiser of *FormalsList* is **true**, then return **true**.
2. Return HasInitialiser of *FormalParameter*.

Static Semantics: IsConstantDeclaration

FunctionDeclaration **:** **function** *BindingIdentifier* **(** *FormalParameters* **)** **{** *FunctionBody* **}**

1. Return **false**.

Static Semantics: IsSimpleParameterList

FormalParameters **:** [empty]

1. Return **true**.

FormalParameterList **:** *FunctionRestParameter*

1. Return **false**.

FormalParameterList **:** *FormalsList* **,** *FunctionRestParameter*

1. Return **false**.

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. If IsSimpleParameterList of *FormalsList* is **false**, return **false**.
2. Return IsSimpleParameterList of *FormalParameter*.

FormalParameter **:** *BindingElement*

1. If HasInitialiser of *BindingElement* is **true**, return **false**.
2. If *FormalParameter* Contains *BindingPattern* is **true**, return **false**.
3. Return **true**.

Static Semantics: IsStrict

FunctionStatementList **:** *StatementList*opt

1. If this *FunctionStatementList* is contained in strict code or if *StatementList* is strict code, then return **true**. Otherwise, return **false**.

Static Semantics: LexicallyDeclaredNames

FunctionDeclaration **:** **function** *BindingIdentifier* **(** *FormalParameters* **)** **{** *FunctionBody* **}**

1. Return the BoundNames of *BindingIdentifier*.

FunctionStatementList **:** [empty]

1. Return an empty List.

FunctionStatementList **:** *StatementList*

1. Return TopLevelLexicallyDeclaredNames of *StatementList*.

Static Semantics: VarDeclaredNames

Deleted: *FormalsList* **:** *FormalParameter* ¶
<#>Return HasInitialiser of *FormalParameter*.

Deleted: *FormalParameter* **:** *BindingElement*
<#>Return HasInitialiser of *BindingElement*.¶

Deleted: *List*

Deleted: *List*

Deleted: *Body*

Deleted: *StatementList*opt

Deleted: *Body*

Deleted:

Comment [AWB10121]: Need a better definition

Deleted: ¶

Deleted: *List*

Deleted: *Body*

Deleted: *Body*

Deleted: *FunctionBody* **:** *StatementList* ¶
<#>Return the LexicallyDeclaredNames of *StatementList*.¶

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. Return an empty List.

FunctionBody **:** [empty]

1. Return an empty List.

FunctionBody **:** *StatementList*

1. Return TopLevelVarDeclaredNames of *StatementList*.

13.1.1.2 Runtime Semantics

Runtime Semantics: Binding Initialisation

With parameters *value* and *environment*.

NOTE When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialised in order to deal with the possibility of multiple parameters with the same name.

FormalParameters **:** [empty]

1. Return NormalCompletion(empty).

FormalParameterList **:** *FunctionRestParameter*

1. Return the result of performing Indexed Binding Initialisation for *FunctionRestParameter* using *value*, 0, and *environment* as the arguments.

FormalParameterList **:** *FormalsList*

1. Return the result of performing Indexed Binding Initialisation for *FormalsList* using *value*, 0, and *environment* as the arguments.

FormalParameterList **:** *FormalsList* **,** *FunctionRestParameter*

1. Let *restIndex* be the result of performing Indexed Binding Initialisation for *FormalsList* using *value*, 0, and *environment* as the arguments.
2. ReturnIfAbrupt(*restIndex*).
3. Return the result of performing Indexed Binding Initialisation for *FunctionRestParameter* using *value*, *restIndex*, and *environment* as the arguments.

Runtime Semantics: EvaluateBody

With parameter *functionObject*.

FunctionBody **:** $FunctionStatementList_{opt}$

1. The code of this *FunctionBody* is strict mode code if it is contained in strict mode code or if the Directive Prologue (14.1) of its *FunctionStatementList* contains a Use Strict Directive or if any of the conditions in 10.1.1 apply. If the code of this *FunctionBody* is strict mode code, *FunctionStatementList* is evaluated in the following steps as strict mode code. Otherwise, *StatementList* is evaluated in the following steps as non-strict mode code.
2. If *FunctionStatementList* is not present, then return NormalCompletion(**undefined**).
3. Let *result* be the result of evaluating *FunctionStatementList*.
4. If *result*.[[type]] is return then return NormalCompletion(*result*.[[value]])
5. ReturnIfAbrupt(*result*).
6. Return NormalCompletion(**undefined**).

Deleted: *List*

Deleted: FunctionBody *:* StatementList ¶ Return the VarDeclaredNames of *StatementList*.¶

Deleted: Initialization

Deleted: *enviornment*

Deleted: and optional parameter *index*

Deleted: initialization

Deleted: the

Deleted: z

Deleted: *List*

Deleted: Perform

Deleted: Initialization

Deleted: *lenValue*,

Deleted: .¶ Return.

Deleted: Asset: *value* is a new created arguments object and hence it has a valid integer valued **`"length"`** property.¶ Let *lenValue* be the result of calling the [[Get]] internal method of *value* with argument **`"length"`**.¶

Deleted: Perform

Deleted: Initialization

Deleted: *lenValue*,

Deleted: Return.¶

Deleted: Initialization

Deleted: *lenValue*,

Deleted: Perform

Deleted: Initialization

Deleted: *lenValue*,

Deleted: .[[value]]

FunctionBody **:** *ThrowTypeError*

1. Throw a **TypeError** exception.

Runtime Semantics: Indexed Binding Initialisation

With parameters *array*, *nextIndex*, and *environment*.

FormalsList **:** *FormalParameter*

1. Let *status* be the result of performing Indexed Binding Initialisation for *FormalParameter* using *array*, *nextIndex*, and *environment* as the arguments.
2. ReturnIfAbrupt(*status*).
3. Return *nextIndex* + 1.

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. Let *lastIndex* be the result of performing Indexed Binding Initialisation for *FormalsList* using *array*, *nextIndex*, and *environment* as the arguments.
2. ReturnIfAbrupt(*lastIndex*).
3. Let *status* be the result of performing Indexed Binding Initialisation for *FormalParameter* using *array*, *lastIndex*, and *environment* as the arguments.
4. ReturnIfAbrupt(*status*).
5. Return *lastIndex* + 1.

FunctionRestParameter **:** **...** *BindingIdentifier*

1. Assert: *array* is a well formed arguments object and hence it has a valid integer valued `"length"` property.
2. Let *status* be the result of Get(*array*, `"length"`).
3. Let *argumentsLength* be *status*.[[value]].
4. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
5. Let *n*=0;
6. Repeat, while *nextIndex* < *argumentsLength*
   a. Let *P* be ToString(*nextIndex*).
   b. Assert: *array* is a well formed arguments object, hence it must have a property *P*.
   c. Let *v* be the result of Get(*array*, *P*).
   d. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*.[[value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
   e. Let *n* = *n*+1.
   f. Let *nextIndex* = *nextIndex* +1.
7. Return the result of performing Binding Initialisation for *BindingIdentifier* using *A* and *environment* as arguments.

Runtime Semantics: InstantiateFunctionObject

With parameter *scope*.

FunctionDeclaration **:** **function** *BindingIdentifier* **(** *FormalParameters* **)** **{** *FunctionBody* **}**

1. If the *FunctionDeclaration* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Let *F* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameters*, *FunctionBody*, *scope*, and *strict*.
3. Perform the abstract operation MakeConstructor with argument *F*.
4. Return *F*.

Deleted: <#>Return.¶
Deleted: Initialization
Deleted: *arrayLength*,
Deleted: Perform
Deleted: Initialization
Deleted: *value*
Deleted: *env*, and
Deleted: *i*
Deleted: *i*
Deleted: Initialization
Deleted: *value*
Deleted: *lenValue*,
Deleted: Perform
Deleted: Initialization
Deleted: *value*
Deleted: *lenValue*,
Deleted: *argumentsLength*
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: *value*
Deleted: with argument
Deleted: *argumentsLength*
Deleted: ion
Deleted: (15.4)
Deleted: a new array object created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name
Deleted: *arrayLength*
Deleted: calling the [[
Deleted:]] internal method of
Deleted: passing
Deleted: as the argument
Deleted: ,
Deleted: , and **false**
Deleted: Perform
Deleted: Initialization
Deleted: *FormalParameter* **:** *BindingElement* ¶ <#>Return the result of performing Perform Indexed Binding InitializationInitialisation for *BindingElement* using *array*, *arrayLength*, *nextIndex*, and *environment* as the arguments.¶
Deleted: Instantiate Function Declaration
Deleted: *List*
Deleted: <#>Let *scope* be the LexicalEnvironment of the running current
Deleted: *List*
Deleted: ing

Runtime Semantics: Evaluation

FunctionDeclaration : **`function`** *BindingIdentifier* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. Return NormalCompletion(empty).

FunctionExpression : **`function`** **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. If the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameters*, *FunctionBody*, *scope*, and *strict*.
4. Perform the abstract operation MakeConstructor with argument *closure*.
5. Return *closure*.

FunctionExpression : **`function`** *BindingIdentifier* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. If the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Let *funcEnv* be the result of calling NewDeclarativeEnvironment passing the running execution context's Lexical Environment as the argument
3. Let *envRec* be *funcEnv's* environment record.
4. Let *name* be StringValue of *BindingIdentifier*.
5. Call the CreateImmutableBinding concrete method of *envRec* passing *name* as the argument.
6. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameters*, *FunctionBody*, *funcEnv*, and *strict*.
7. Perform the abstract operation MakeConstructor with argument *closure*.
8. Call the InitialiseBinding concrete method of *envRec* passing *name* and *closure* as the arguments.
9. Return NormalCompletion(*closure*).

NOTE 1 The *BindingIdentifier* in a *FunctionExpression* can be referenced from inside the *FunctionExpression's* *FunctionBody* to allow the function to call itself recursively. However, unlike in a *FunctionDeclaration*, the *BindingIdentifier* in a *FunctionExpression* cannot be referenced from and does not affect the scope enclosing the *FunctionExpression*.

NOTE 2 A **`prototype`** property is automatically created for every function defined using a *FunctionDeclaration* or *FunctionExpression*, to allow for the possibility that the function will be used as a constructor.

13.2 Arrow Function Definitions

Syntax

ArrowFunction **:**
  ArrowParameters **`=>`** *ConciseBody*

ArrowParameters **:**
  BindingIdentifier
  CoverParenthesisedExpressionAndArrowParameterList

ConciseBody **:**
  [lookahead ∉ { **`{`** }] *AssignmentExpression*
  `{` *FunctionBody* **`}`**

Supplemental Syntax

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

ArrowFormalParameters **:**
  `(` *StrictFormalParameters* **`)`**

Deleted: The production
Deleted: *List*
Deleted: opt
Deleted: .
is instantiated as follows during Declaration Binding instantiation (10.5):
Deleted: (normal, empty, empty).
Deleted: Return the result of creating a new Function obj as specified in 13.2 with parameters specified by *FormalParameterList*opt, and body specified by *FunctionB* Pass in the VariableEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *FunctionDeclaration* is contained in strict code or if its *FunctionBody* is strict code.
Deleted: The production
Deleted: *List*
Deleted: opt
Deleted: .
is evaluated as follows:
Deleted: current
Deleted: *List*
Deleted: ing
Deleted: the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList*opt and body specified by *FunctionBo* Pass in the LexicalEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code
Deleted: *List*
Deleted: opt
Deleted: running
Deleted: current
Deleted:
Deleted: the String value of *BindingIdentifier*
Deleted: <#>Let *closure* be the result of creating a new
Deleted: *FormalParameterList*
Deleted: ing
Deleted: ize
Deleted: Immutable
Deleted: the String value of *Identifier*
Deleted: Value
Deleted: The production *FunctionBody* **:**
Deleted: *z*
Deleted: **`(`** *ArrowFormalParameterList* **`)`**
Deleted: *ArrowFormalParameterList* **:**¶
Deleted: *z*
Deleted: *z*
Deleted: *List*
Deleted: *List*

13.2.1.1 Static Semantics

Static Semantics: Early Errors

ArrowFunction : *ArrowParameters* => *ConciseBody*

- It is a Syntax Error if any element of the BoundNames of *ArrowParameters* also occurs in the VarDeclaredNames of *ConciseBody*.
- It is a Syntax Error if any element of the BoundNames of *ArrowParameters* also occurs in the LexicallyDeclaredNames of *ConciseBody*.

ArrowParameters : *BindingIdentifier*

- It is a Syntax Error if the StringValue of the sole element of the BoundNames of *BindingIdentifier* is **`eval`** or **`arguments`**.

ArrowParameters : *CoverParenthesisedExpressionAndArrowParameterList*

- It is a Syntax Error if the lexical token sequence matched by *CoverParenthesisedExpressionAndArrowParameterList* cannot be parsed with no tokens left over using *ArrowFormalParameters* as the goal symbol.
- It is a Syntax Error if any early errors are present for CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.

ConciseBody : [lookahead ∉ { **{** }] *AssignmentExpression*

- It is a Syntax Error if *AssignmentExpression* Contains *YieldExpression*.

Static Semantics: BoundNames

ArrowParameters : *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return the BoundNames of *formals*.

Static Semantics: Contains

With parameter *symbol*.

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. If *symbol* is neither **`super`** or **`this`**, then return **false**.
2. If *ArrowParameters* Contains *symbol* is **true**, return **true**;
3. Return *ConciseBody* Contains *symbol* .

NOTE Normally, Contains does not look inside most function forms. However, Contains is used to detect **`this`** and **`super`** usage within an *ArrowFunction*.

ArrowParameters : *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return *formals* Contains *symbol*.

Static Semantics: CoveredFormalsList

ArrowParameters : *BindingIdentifier*

Deleted: <#>CoverFormalsList :¶
<#>Expression¶
<#>Supplemental Syntax¶
<#>When processing the production *CoverFormalsList* **:** *Expression* **the** *FormalsList* **production is used to furthe restrict the source code that matches** *Expression***.¶**
<#>ArrowFormalParameterList :¶
<#>FormalParameterList ¶

Deleted: *ConciseBody*

Deleted: IsSimpleParameterList of *ArrowParameters* is **true** and

Deleted: *FormalParameterList*

Deleted: <#>It is a Syntax Error if IsSimpleParameterList of *ArrowParameters* **false** and BoundNames of *ArrowParameters* contains any duplicate elements.¶
<#>It is a Syntax Error if IsSimpleParameterList of *ArrowParameters* **false** and BoundNames of *ArrowParameters* contains either **"eval"** or **"arguments"**.¶
<#>It is a Syntax Error if the source code matching this production is strict code and

Deleted: <#>It is a Syntax Error if any

Deleted: *ConciseBody*

Deleted: <#>It is a Syntax Error if

Deleted:

Deleted: *ArrowFormalParameterList* **:**

Deleted: *CoverFormalsList*

Deleted: *z*

Deleted: *Expression*

Deleted: source code

Deleted: corresponding to

Deleted: *z*

Deleted: *Expression*

Deleted: *List*

Deleted: *FormalsList*

Deleted: <#>All Early Errors rules for

Deleted: ¶

Deleted: *z*

Deleted: *z*

Deleted: *ArrowParameters* **:** *BindingIdentifer*¶

Deleted: *ArrowFormalParameterList* **:** [empty]

Deleted: *ConsiceBody*

Deleted: s

Deleted: s

Deleted: *z*

Deleted: *z*

Deleted: NOTE Static semantic rules that dep

Deleted:

1. Return *BindingIdentifier*.

CoverParenthesisedExpressionAndArrowParameterList **:**
  (*Expression* **)**
  ( )
  (... *Identifier* **)**
  (*Expression* **, ...** *Identifier* **)**

1. Return the result of parsing the lexical token stream matched by *CoverParenthesisedExpressionAndArrowParameterList* using *ArrowFormalParameters* as the goal symbol.

Static Semantics: ExpectedArgumentCount

ArrowParameters **:** *BindingIdentifier*

2. Return 1.

ArrowParameters **:** *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return the ExpectedArgumentCount of *formals*.

Static Semantics: IsSimpleParameterList

ArrowParameters **:** *BindingIdentifier*

1. Return **true**.

ArrowParameters **:** *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return the IsSimpleParameterList of *formals*.

Static Semantics: LexicallyDeclaredNames

ConciseBody **:** [lookahead ∉ { **{** }] *AssignmentExpression*

1. Return an empty List.

13.2.1.2 **Runtime Semantics**

Runtime Semantics: Binding Initialisation

With parameters *value* and *environment*.

NOTE When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialised in order to deal with the possibility of multiple parameters with the same name.

ArrowParameters **:** *BindingIdentifier*

1. Return the result of performing Binding Initialisation for *BindingIdentifier* using *value* and *environment* as the arguments.

ArrowParameters **:** *CoverParenthesisedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesisedExpressionAndArrowParameterList*.
2. Return the result of performing Binding Initialisation of *formals* with arguments *value* and *environment*.

Deleted: *z*

Deleted: *CoverFormalsList* **:** *Expression*¶

Deleted: source code

Deleted: corresponding to

Deleted: *z*

Deleted: *Expression*

Deleted: *List*

Deleted: *FormalsList*

Deleted: *ArrowParameters* **:** **(** *ArrowFormalParameterLi* **)** ¶
<#>Return the ExpectedArgumentCount of *ArrowFormalParameterList*.¶

Deleted: *ArrowFormalParameterList* **:**¶
[empty]
FunctionRestParameter¶
<#>Return 0.¶
ArrowFormalParameterList **:**¶
CoverFormalsList ,
CoverFormalsList **,** *FunctionRestParameter*¶

Deleted: *z*

Deleted: *z*

Deleted: CoveredCFormalsList of *CoverFormalsList*

Deleted: *z*

Deleted: *z*

Deleted: *ConsiceBody*

Deleted: <#>ConsiceBody : { **FunctionBody**
<#>Return the LexicallyDeclaredNames of *FunctionBody*.¶
<#>Static Semantics: VarDeclaredNames¶
<#>ConsiseBody : *[lookahead ∉ { { }]* AssignmentExpression ¶
<#>Return the VarDeclaredNames of *AssignmentExpression*.¶
<#>ConsiseBody : { **FunctionBody** }¶
<#>Return the VarDeclaredNames of *FunctionBody*.¶

Deleted: and optional parameter *index*

Deleted: z

Deleted: Indexed

Deleted: , *lenValue*

Deleted: ,

Deleted: 0,

Deleted: *z*

Deleted: *z*

Deleted: ro

Deleted: i

Deleted: *z*

Deleted: and optional argument *index*, if index was pass to this routine

Runtime Semantics: EvaluateBody

With parameter *functionObject*.

ConciseBody **:** [lookahead ∉ { **{** }] *AssignmentExpression*

1. The code of this *ConciseBody* is strict mode code if it is contained in strict mode code or if any of the conditions in 10.1.1 apply. If the code of this *ConciseBody* is strict mode code, *AssignmentExpression* is evaluated in the following steps as strict mode code. Otherwise, *AssignmentExpression* is evaluated in the following steps as non-strict mode code.
2. Let *exprRef* be the result of evaluating *AssignmentExpression*.
3. Let *exprValue* be GetValue(*exprRef*).
4. If *exprValue*.[[type]] is return then return NormalCompletion(*exprValue*.[[value]]).
5. ReturnIfAbrupt(*exprValue*).
6. Return NormalCompletion(*exprValue*).

NOTE In the absence of extensions to this specification, the test is step 4 will never be **true**.

Runtime Semantics: Evaluation

ArrowFunction **:** *ArrowParameters* **=>** *ConciseBody*

1. Let *strict* be **true**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *parameters* be CoveredFormalsList of *ArrowParameters*.
4. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Arrow, *parameters*, *ConciseBody*, *scope*, and *strict*.
5. Return *closure*.

NOTE Even though an *ArrowFunction* may contain references to **`super`**, the FunctionCreate call in step 3 is not passed the optional *homeObject* and *methodName* parameters. An *ArrowFunction* that references **`super`** is always contained within a non-*ArrowFunction* and the necessary state to implement **`super`** is accessible via the *scope* that is captured by the function object of the *ArrowFunction*.

13.3 Method Definitions

Syntax

MethodDefinition **:**
PropertyName **(** *StrictFormalParameters* **)** **{** *FunctionBody* **}**
GeneratorMethod
`get` *PropertyName* **( )** **{** *FunctionBody* **}**
`set` *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

PropertySetParameterList **:**
BindingIdentifier
BindingPattern

NOTE The single element of a *PropertySetParameterList* may not have a default value *Initialiser* because set accessor are always called with an implicitly provided argument.

13.3.1.1 Static Semantics

Static Semantics: Early Errors

MethodDefinition **:** *PropertyName* **(** *StrictFormalParameters* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if any element of the BoundNames of *StrictFormalParameters* also occurs in the VarDeclaredNames of *FunctionBody*.

Comment [AWB7122]: TODO, need to resolve whether or not ArrowBodies are alwa strict

Comment [AWB9123]: TODO If exprRef is Reference that invokes a getter we probably should find a way to specify that the get call i handled as a tail call

Deleted: *ArrowParameters* **: (** *ArrowFormalParameterList* **)** ¶
<#>Return the result of performing Binding Initialisation for *ArrowFormalParameterList* us *value*, *environment* as the arguments. Also pass optional argument *index* if it is present..¶

Deleted: *ArrowFormalParameterList* **:** [empty]
<#>Return NormalCompletion(empty).¶
ArrowFormalParameterList **:** *FunctionRestParameter*¶
<#>Return the result of performing Indexed Binding Initialisation for *FunctionRestParamete* using *value*, *lenValue*, 0, and *environment* as the arguments..¶
ArrowFormalParameterList **:** *CoverFormalsList*¶
<#>Return the result of performing Indexed Binding Initialisation for CoveredCFormalsList *CoverFormalsList* using *value*, *lenValue*, 0, and *environment* as the arguments.¶

Deleted: *ConsiseBody*

Comment [AWB7124]: Confirm: concise

Deleted: current

Deleted: *ArrowParameterLists*

Comment [AWB7125]: TODO, need to

Comment [AWB9126]: TODO If exprRef is

Deleted: *ConsiseBodyConciseBody* **:** [lookahe

Deleted: <#>ConsiseBody *:*

Deleted: *List*

Deleted: ConsiseBody

Deleted: ***** *PropertyName*

Deleted: **(** *FormalParameterList* **)**

Deleted: **{**

Deleted: ConsiseBody

Deleted: *FunctionBody* **}**

Deleted:

Deleted: **{**

Deleted: ConsiseBody

Deleted: *FunctionBody* **}**

Deleted: ed

Deleted: *MethodDefinition* **:** *PropertyName*

Deleted: <#>It is a Syntax Error if any

Deleted: *F*

Deleted: *List*

Deleted: *ConsiseBody*

Deleted:

- It is a Syntax Error if any element of the BoundNames of *StrictFormalParameters* also occurs in the LexicallyDeclaredNames of *FunctionBody*.

MethodDefinition **:** **`set`** *PropertyName* **`(`** *PropertySetParameterList* **`)`** **`{`** *FunctionBody* **`}`**

- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **false** and any element of the BoundNames of *PropertySetParameterList* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **false** and BoundNames of *PropertySetParameterList* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **false** and BoundNames of *PropertySetParameterList* contains either **`"eval"`** or **`"arguments"`**.
- It is a Syntax Error if BoundNames of *PropertySetParameterList* contains any duplicate elements.
- It is a Syntax Error if any element of the BoundNames of *PropertySetParameterList* also occurs in the LexicallyDeclaredNames of *FunctionBody*.
- It is a Syntax Error if *PropertySetParameterList* Contains *YieldExpression*.

Static Semantics: ExpectedArgumentCount

PropertySetParameterList **:** *BindingIdentifier*

1. Return 1.

PropertySetParameterList **:** *BindingPattern*

1. Return 1.

Static Semantics: IsSimpleParameterList

PropertySetParameterList **:** *BindingIdentifier*

1. Return **true**.

PropertySetParameterList **:** *BindingPattern*

1. Return **false**.

Static Semantics: PropName

MethodDefinition **:**
PropertyName **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**
`get` *PropertyName* **`(`** **`)`** **`{`** *FunctionBody* **`}`**
`set` *PropertyName* **`(`** *PropertySetParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return PropName of *PropertyName*.

Static Semantics: ReferencesSuper

MethodDefinition **:** *PropertyName* **`(`** *FormalParameters* **`)`** **`{`** *FunctionBody* **`}`**

1. If *FormalParameters* Contains **`super`** is **true**, then return **true**.
2. Return *FunctionBody* Contains **`super`**.

MethodDefinition **:** **`get`** *PropertyName* **`(`** **`)`** **`{`** *FunctionBody* **`}`**

1. Return *FunctionBody* Contains **`super`**.

MethodDefinition **:** **`set`** *PropertyName* **`(`** *PropertySetParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. If *PropertySetParameterList* Contains **`super`** is **true**, then return **true**.

Deleted: <#>It is a Syntax Error if *ConsiceBody FunctionBody* Contains *YieldExpression*.¶ *MethodDefinition* **:** **`*`** *PropertyName* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**¶ <#>It is a Syntax Error if *FunctionBody* Contains *YieldExpression* is **false**.¶ *MethodDefinition* **:** **`get`** *PropertyName* **`( )`** ***ConsiseBoc*** *FunctionBody* **`}`** ¶ <#>It is a Syntax Error if *ConsiceBodyFunctionBody* Contains *YieldExpression*.¶

Deleted: *ConsiseBody*

Deleted: *FormalParameterList*

Deleted: **true**

Deleted: *FormalParameterList*

Deleted: *FormalParameterList*

Deleted: *FormalsParameterList*

Deleted: *FormalParameterList*

Deleted: *FormalParameterList*

Deleted: *ConciseBody*

Deleted: <#>It is a Syntax Error if *ConsiceBodyFunctionBody* Contains *YieldExpression*.¶

Deleted: *PropertySetParameterList* **:** *BindingPattern* ¶ <#>It is a Syntax Error if BoundNames of *BindingPatt* contains any duplicate elements.¶

Deleted: Static Semantics: BoundNames¶ *PropertySetParameterList* **:** *BindingIdentifier* ¶ <#>Return BoundNames of *BindingIdentifier*.¶ *PropertySetParameterList*: *BindingPattern* ¶ <#>Return BoundNames of *BindingPattern*.¶

Deleted: *List*

Deleted: **`*`** *PropertyName* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

Deleted: *MethodDefinition* **:** *PropertyName* **`(`** *FormalParameterList* **`)`** *ConsiseBody* ¶

Deleted: *MethodDefinition* **:** **`*`** *PropertyName* **`(`** *FormalParameterList* **`)`** *ConsiseBody* ¶ <#>Return PropName of *PropertyName*.¶ *MethodDefinition* **:** **`get`** *PropertyName* **`( )`** ***ConsiseBo*** <#>Return PropName of *PropertyName*.¶ *MethodDefinition* **:** **`set`** *PropertyName* **`(`** *PropertySetParameterList* **`)`** *ConsiseBody* ¶ <#>Return PropName of *PropertyName*.¶

Deleted: ¶

Deleted: *List*

Deleted: **`}`** **`*`** *PropertyName* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBo* **`}`**

Deleted: **`get`** *PropertyName* **`( )`** **`{`** *FunctionBody* **`}`** **`set`** *PropertyName* **`(`** *PropertySetParameterList* **`)`** **`{`** *FunctionBody* **`}`**

Deleted: *List*

2. Return *FunctionBody* Contains **super**.

Static Semantics: SpecialMethod

MethodDefinition **:** *PropertyName* **(** *StrictFormalParameters* **)** **{** *FunctionBody* **}**

1. Return **false**.

MethodDefinition **:**
GeneratorMethod
get *PropertyName* **( )** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

1. Return **true**.

13.3.1.2 Runtime Semantics

Runtime Semantics: Property Definition Evaluation

With parameter *object* and optional parameter *functionPrototype*.

MethodDefinition **:** *PropertyName* **(** *StrictFormalParameters* **)** **{** *FunctionBody* **}**

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *strict* be IsStrict of *FunctionBody*.
4. Let *scope* be the running execution context's LexicalEnvironment.
5. If ReferencesSuper of *MethodDefinition* is **true**, then
   a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *StrictFormalParameters*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument. If *functionPrototype* was passed as a parameter then also pass its value as the *functionPrototype* optional argument of FunctionCreate.
6. Else
   a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *StrictFormalParameters*, *FunctionBody*, *scope*, and *strict*. If *functionPrototype* was passed as a parameter then also pass its value as the *functionPrototype* optional argument of FunctionCreate.
7. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. Let *status* be the result of DefinePropertyOrThrow(*object*, *propName*, *desc*).
9. ReturnIfAbrupt(*status*).
10. NormalCompletion(closure).

MethodDefinition **:** *GeneratorMethod*

See 13.4.

MethodDefinition **:** **get** *PropertyName* **( )** **{** *FunctionBody* **}**

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *strict* be IsStrict of *FunctionBody*.
4. Let *scope* be the running execution context's LexicalEnvironment.
5. Let *formalParameterList* be the production *FormalParameters* **:** [empty]
6. If ReferencesSuper of *MethodDefinition* is **true**, then
   a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *formalParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else

Deleted: *FormalParameterList*
Deleted: *
Deleted: *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
Deleted: .
Deleted: *FormalParameterList*
Deleted: *ConsiseBody*
Deleted:
Deleted: PropName of
Deleted: true
Deleted: current
Deleted: <#>Let *needsSuperBinding* be the result of *FunctionBody* Contains **super**.¶
<#>If *needsSuperBinding* is **false**, then let *needsSuperBinding* be the result of *FormalParameterList* Contains **super**.¶
Deleted: *needsSuperBinding*
Deleted: *FormalParameterList*
Deleted: *ConciseBody*
Deleted: *FormalParameterList*
Comment [AWB10127]: At its sept 2012 meeting TC39 decided that it wants methods
Deleted: **true**,
Deleted: **false**
Deleted: **false**
Deleted: Return
Deleted: calling the [[
Deleted: Own
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: ,
Deleted: and
Deleted: , and **false**
Deleted: *MethodDefinition*
Deleted: **:** * *PropertyName*
Deleted:
Comment [AWB8128]: A default value
Deleted: <#>Let *propName* be PropName of
Deleted: *ConsiseBody*
Deleted: **{** *FunctionBody* **}**
Deleted: PropName of
Deleted: If this *MethodDefinition* is contained
Deleted: current
Deleted: *List*
Deleted: <#>Let *needsSuperBinding* be the
Deleted: *needsSuperBinding*

a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *formalParameterList*, *FunctionBody*, *scope*, and *strict*.

8. Let *desc* be the Property Descriptor {[[Get]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
9. Let *status* be the result of DefinePropertyOrThrow(*object*, *propName*, *desc*).
10. ReturnIfAbrupt(*status*).
11. Return NormalCompletion(*closure*).

MethodDefinition **:** `set` *PropertyName* `(` *PropertySetParameterList* `)` `{` *FunctionBody* `}`

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *strict* be IsStrict of *FunctionBody*.
4. Let *scope* be the running execution context's LexicalEnvironment.
5. If ReferencesSuper of *MethodDefinition* is **true**, then
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *PropertySetParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
6. Else
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *PropertySetParameterList*, *FunctionBody*, *scope*, and *strict*.
7. Let *desc* be the Property Descriptor {[[Set]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
8. Let *status* be the result of DefinePropertyOrThrow(*object*, *propName*, *desc*).
9. ReturnIfAbrupt(*status*).
10. Return NormalCompletion(*closure*).

13.4 Generator Function Definitions

Syntax

GeneratorMethod **:**
    `*` *PropertyName* `(`*StrictFormalParameters*`)` `{` *FunctionBody* `}`

GeneratorDeclaration **:**
    `function` `*` *BindingIdentifier* `(` *FormalParameters* `)` `{` *FunctionBody* `}`

GeneratorExpression **:**
    `function` `*` *BindingIdentifier*$_{opt}$ `(` *FormalParameters* `)` `{` *FunctionBody* `}`

YieldExpression **:**
    `yield` *YieldDelegator*$_{opt}$ [Lexical goal *InputElementRegExp*] *AssignmentExpression*

YieldDelegator **:**
    `*`

Supplemental Syntax

The following productions are used as an aid in specifying the semantics of certain ECMAScript language features. They are not used when parsing ECMAScript source code.

GeneratorBody **:**
    FunctionBody
    Comprehension

13.4.1.1 Static Semantics

Static Semantics: Early Errors

GeneratorMethod **:** `*` *PropertyName* `(`*StrictFormalParameters* `)` `{` *FunctionBody* `}`

Deleted: <#>Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method *formalParameterList*, *ConciseBody*, *scope*, and *strict* and *object* as the *homeObject* optional argument and *propName* the *methodName* optional argument.¶

Deleted: <#>TODO: Make closure a constructor, for ES compat¶

Deleted: Return

Deleted: calling the [[DefineOwnProperty]] internal met of *object* with arguments *propName*, and *desc*

Deleted: , and **false**

Deleted: *o*

Deleted: *ConsiseBody*

Deleted: `{` *FunctionBody* `}`

Deleted: PropName of

Deleted: If this *MethodDefinition* is contained in strict c or if *FunctionBody ConciseBody* is strict code, then let *str* be **true**. Otherwise, let *strict* be **false**.

Deleted: current

Deleted: <#>Let *needsSuperBinding* be the result of *FunctionBody* Contains **`super`**.¶
<#>If *needsSuperBinding* is **false**, then let *needsSuperBinding* be the result of *PropertySetParameter*l cContains **`super`**.¶

Deleted: *needsSuperBinding*

Deleted: *o*

Deleted: *o*

Deleted: <#>Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method *PropertySetParameterList*, *ConciseBody*, *scope*, and *stric* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.¶

Deleted: <#>TODO: Make closure a constructor, for ES compat¶

Deleted: Return

Deleted: calling the [[

Deleted: Own

Deleted:]] internal method of

Deleted: with arguments

Deleted: ,

Deleted: and

Deleted: , and **false**

Deleted: <#>NOTE The strictness and propert attributes for `set` and `get` *MethodDefinition* productions differ from the other *MethodDefinition*

Deleted: f

Deleted: *List*

Deleted: *Function*

Deleted: *List*

Deleted: *Function*

Comment [AWB10129]: This actually doesn't

Deleted: *GeneratorMethod* **:** `*` *PropertyName*

- It is a Syntax Error if any element of the BoundNames of *StrictFormalParameters* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if any element of the BoundNames of *StrictFormalParameters* also occurs in the LexicallyDeclaredNames of *FunctionBody*.

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }
and
GeneratorExpression : **function** * *BindingIdentifier*$_{opt}$ (*FormalParameters*) { *FunctionBody* }

- If the source code matching this production is strict code, the Early Error rules for *StrictFormalParameters* : *FormalParameters* are applied.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameters* is **false** and any element of the BoundNames of *FormalParameters* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if any element of the BoundNames of *FormalParameters* also occurs in the LexicallyDeclaredNames of *FunctionBody*.

Static Semantics: BoundNames

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: Contains

With parameter *symbol*.

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return **false**.

GeneratorExpression : **function** * *BindingIdentifier*$_{opt}$ (*FormalParameters*) { *FunctionBody* }

1. Return **false**.

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

Static Semantics: IsConstantDeclaration

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return **false**.

Static Semantics: LexicallyDeclaredNames

GeneratorDeclaration : **function** * *BindingIdentifier* (*FormalParameters*) { *FunctionBody* }

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: PropName

GeneratorMethod : * *PropertyName* (*FormalParameters*) { *FunctionBody* }

1. Return PropName of *PropertyName*.

Static Semantics: ReferencesSuper

GeneratorMethod : * *PropertyName* (*FormalParameters*) { *FunctionBody* }

1. If *FormalParameters* Contains **super** is **true**, then return **true**.

Deleted: *YieldExpression* : **yield** *YieldDelegator*$_{opt}$ *AssignmentExpression*) ¶ <#>It is a Syntax Error if any *AssignmentExpression* Contains *YieldExpressi*

Deleted: *List*

Deleted: *FunctionBody*

Deleted: *FunctionDeclaration*

Deleted: *List*

Deleted: *FunctionBody*

Deleted: *Function*

Deleted: *List*

Deleted: *FunctionBody*

Deleted: *List*

Deleted: *FunctionBody*

Deleted: *List*

Deleted: *FunctionBody*

2. Return *FunctionBody* Contains **super**.

Static Semantics: VarDeclaredNames

GeneratorDeclaration : **function *** *BindingIdentifier* **(** *FormalParameters* **) {** *FunctionBody* **}**

1. Return an empty List.

13.4.1.2 Runtime Semantics

NOTE: Abstract operations relating to generator objects are defined in 15.19.4.3.

Runtime Semantics: EvaluateBody

With parameter *functionObject*.

GeneratorBody : *FunctionBody*

1. Assert: A Function Environment Record containing a this binding has already been activated as the current environment.
2. Let *env* be the result of the GetThisEnvironment abstract operation.
3. Let *G* be the of calling the result of calling the GetThisBinding concrete method of *env*.
4. If Type(*G*) is not Object or if Type(*G*) is Object and *G* does not have a [[GeneratorState]] internal data property or if Type(*G*) is Object and *G* has a [[GeneratorState]] internal data property and the value of *G's* [[GeneratorState]] internal data property is not **undefined**, then
    a. Let *newG* be the result of calling OrdinaryCreateFromConstructor(*functionObject*, **"%GeneratorPrototype%"**, ([[GeneratorState]], [[GeneratorContext]])).
    b. ReturnIfAbrupt(*newG*).
    c. Let *G* be *newG*.
5. Return the result of GeneratorStart(*G*, *FunctionBody*).

GeneratorBody : *Comprehension*

1. Let *G* be the result of ObjectCreate(**"%GeneratorPrototype%"**, ([[GeneratorState]], [[GeneratorContext]])).
2. ReturnIfAbrupt(*G*).
3. Assert: the value of *G's* [[GeneratorState]] internal data property is **undefined**.
4. Let *startStatus be* the result of GeneratorStart(*G*, *Comprehension*).
5. ReturnIfAbrupt(*startStatus*).
6. Return *G*.

Runtime Semantics: InstantiateFunctionObject

With parameter *scope*.

GeneratorDeclaration : **function *** *BindingIdentifier* **(** *FormalParameters* **) {** *FunctionBody* **}**

1. If the *GeneratorDeclaration* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Using *FunctionBody* from the production that is being evaluated, let *body* be the supplemental syntactic grammar production: *GeneratorBody* : *FunctionBody*.
3. Let *F* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Normal, *FormalParameters*, *body*, *scope*, and *strict*.
4. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
5. Perform the abstract operation MakeConstructor with arguments *F*, **true**, and *prototype*.
6. Return *F*.

Deleted: *List*

Deleted: *FunctionBody*

Deleted: 15.19.4.3

Deleted: If

Deleted:]]

Deleted: not

Deleted: ,

Deleted: then throw a **TypeError** exception

Runtime Semantics: Property Definition Evaluation

With parameter *object* and optional parameter *functionPrototype*.

GeneratorMethod **:** `*` *PropertyName* `(` *StrictFormalParameters* `)` `{` *FunctionBody* `}`

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *strict* be IsStrict of *FunctionBody*.
4. Let *scope* be the running execution context's LexicalEnvironment.
5. Using *FunctionBody* from the production that is being evaluated, let *body* be the supplemental syntactic grammar production: *GeneratorBody* **:** *FunctionBody*.
6. If ReferencesSuper of *GeneratorMethod* is **true**, then
   a. Let *closure* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Method, *StrictormalParameters*, *body*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else
   a. Let *closure* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Method, *StrictFormalParameters*, *body*, *scope*, and *strict*.
8. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
9. Perform the abstract operation MakeConstructor with arguments *closure*, **true**, and *prototype*.
10. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
11. Let *status* be the result of DefinePropertyOrThrow(*object*, *propName desc*).
12. ReturnIfAbrupt(*status*).
13. Return NormalCompletion(*closure*).

Deleted: PropName of

Runtime Semantics: Evaluation

GeneratorDeclaration **:** `function` `*` *BindingIdentifier* `(` *FormalParameters* `)` `{` *FunctionBody* `}`

1. Return NormalCompletion(empty)

GeneratorExpression **:** `function` `*` `(` *FormalParameters* `)` `{` *FunctionBody* `}`

1. If the *GeneratorExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Using *FunctionBody* from the production that is being evaluated, let *body* be the supplemental syntactic grammar production: *GeneratorBody* **:** *FunctionBody*.
3. Let *scope* be the LexicalEnvironment of the running execution context.
4. Let *closure* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Normal, *FormalParameters*, *body*, *scope*, and *strict*.
5. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
6. Perform the abstract operation MakeConstructor with arguments *closure*, **true**, and *prototype*.
7. Return *closure*.

GeneratorExpression **:** `function` `*` *BindingIdentifier* `(` *FormalParameters*`)` `{` *FunctionBody* `}`

1. If the *GeneratorExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
2. Using *FunctionBody* from the production that is being evaluated, let *body* be the supplemental syntactic grammar production: *GeneratorBody* **:** *FunctionBody*.
3. Let *funcEnv* be the result of calling NewDeclarativeEnvironment passing the running execution context's Lexical Environment as the argument
4. Let *envRec* be *funcEnv's* environment record.
5. Let *name* be StringValue of *BindingIdentifier*.
6. Call the CreateImmutableBinding concrete method of *envRec* passing *name* as the argument.

7. Let *closure* be the result of performing the GeneratorFunctionCreate abstract operation with arguments Normal, *FormalParameters*, *body*, *funcEnv*, and *strict*.
8. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
9. Perform the abstract operation MakeConstructor with arguments *closure*, **true**, and *prototype*.
10. Call the InitialiseBinding concrete method of *envRec* passing *name* and *closure* as the arguments.
11. Return NormalCompletion(*closure*).

NOTE 1 The *BindingIdentifier* in a *GeneratorExpression* can be referenced from inside the *GeneratorExpression's* *FunctionBody* to allow the generator code to call itself recursively. However, unlike in a *GeneratorDeclaration*, the *BindingIdentifier* in a *GeneratorExpression* cannot be referenced from and does not affect the scope enclosing the *GeneratorExpression*.

YieldExpression **:** **`yield`** *YieldDelegator*$_{opt}$ *AssignmentExpression*

1. Let *exprRef* be the result of evaluating *AssignmentExpression*.
2. Let *value* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*value*).
4. If *YieldDelegator* is present, then
    a. Let *iterator* be the result of GetIterator(*value*).
    b. ReturnIfAbrupt(*iterator*).
    c. Let *received* be **undefined**.
    d. Repeat
        i. Let *innerResult* be the result of IteratorNext(*iterator*, *received*).
        ii. ReturnIfAbrupt(*innerResult*).
        iii. Let *done* be IteratorComplete(*innerResult*).
        iv. ReturnIfAbrupt(*done*).
        v. If *done* is **true**, then
            1. Let *innerValue* be the result of IteratorValue (*innerResult*).
            2. Return *innerValue*.
        vi. Let *yieldCompletion* be the result of GeneratorYield(*innerResult*).
        vii. Let *received* be *yieldCompletion*.[[value]].
        viii. If *yieldCompletion*.[[type]] is throw, then
            1. If HasProperty(*iterator*, **`"throw"`**) is **true**, then
                a Let *innerResult* be the result of Invoke(*iterator*, **`"throw"`**, (*received*)).
                b ReturnIfAbrupt(*innerResult*).
            2. Return *yieldCompletion*.
5. Return the result of GeneratorYield(CreateItrResultObject(*value*, **false**)).

Deleted: <#>If Type(*value*) is not then throw a **TypeError** exception.¶
Deleted: performing Invoke with arguments
Deleted:
Deleted: ,
Deleted: @@iterator and an empty List
Deleted: <#>If Type(*iterator*) is not then throw a **TypeError** exception.¶
Deleted: Invoke
Deleted: `"value"`, (
Deleted:)
Deleted: <#>If Type(*innerResult*) is not Object, th throw a **TypeError** exception.¶
Deleted: Get
Deleted: , `"value"`

13.5 Class Definitions

Syntax

ClassDeclaration **:**
    `class` *BindingIdentifier* *ClassTail*

ClassExpression **:**
    `class` *BindingIdentifier*$_{opt}$ *ClassTail*

ClassTail **:**
    ClassHeritage$_{opt}$ **`{`** *ClassBody*$_{opt}$ **`}`**

ClassHeritage **:**
    `extends` *AssignmentExpression*

ClassBody **:**
    ClassElementList

ClassElementList **:**
ClassElement
ClassElementList ClassElement

ClassElement **:**
MethodDefinition
`static` *MethodDefinition*
;

NOTE A *ClassBody* is always strict code.

13.5.1.1 Static Semantics

Static Semantics: Early Errors

ClassBody **:** *ClassElementList*

- It is a Syntax Error if PrototypePropertyNameList of *ClassElementList* contains any duplicate entries, unless the following condition is true for each duplicate entry: The duplicated entry occurs exactly twice in the list and one occurrence was obtained from a **`get`** accessor *MethodDefinition* and the other occurrence was obtained from a **`set`** accessor *MethodDefinition*.
- It is a Syntax Error if StaticPropertyNameList of *ClassElementList* contains any duplicate entries, unless the following condition is true for each duplicate entry: The duplicated entry occurs exactly twice in the list and one occurrence was obtained from a **`get`** accessor *MethodDefinition* and the other occurrence was obtained from a **`set`** accessor *MethodDefinition*.

ClassElement **:** *MethodDefinition*

- It is a Syntax Error if PropName of *MethodDefinition* is **`"constructor"`** and SpecialMethod of *MethodDefinition* is **true**.

ClassElement **:** **`static`** *MethodDefinition*

- It is a Syntax Error if PropName of *MethodDefinition* is **`"prototype"`**.

Static Semantics: BoundNames

ClassDeclaration: **`class`** *BindingIdentifier ClassTail*

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: ConstructorMethod

ClassBody : *ClassElementList*

1. Let *list* be PrototypeMethodDefinitions of *ClassElementList*.
2. For each *MethodDefinition m* in *list*, do
   a. If PropName of *m* is **`"constructor"`**, return *m*.
3. Return empty.

NOTE Early Error rules ensure that there is only one method definition named **`"constructor"`** and that it isn't an accessor property or generator definition.

Static Semantics: Contains

With parameter *symbol*.

ClassTail **:** *ClassHeritage*$_{opt}$ **{** *ClassBody* **}**

Deleted: *ClassDeclaration* **: `class`** *BindingIdentifier ClassTail*
and
ClassExpression **: `class`** *BindingIdentifier ClassTail* ¶
<#>It is a Syntax Error if BoundNames of *BindingIdentifier* contains either **`"eval"`** or **`"arguments"`**¶

Deleted: {

Deleted: }

Deleted: Method

Deleted: *ClassHeritage* **: `extends`** *AssignmentExpression* ¶
<#>It is a Syntax Error if *AssignmentExpression* Contains *YieldExpressi*

1. If *symbol* is *ClassBody*, return **true**.
2. If *ClassHeritage* is not present, return **false**.
3. If *symbol* is *ClassHeritage*, return **true**.
4. Return the result of Contains for *ClassHeritage* with argument *symbol*.

NOTE Static semantic rules that depend upon substructure generally do not look into class bodies.

Static Semantics: IsConstantDeclaration

ClassDeclaration: `class` *BindingIdentifier* *ClassTail*

1. Return **false**.

Static Semantics: IsStatic

ClassElement : *MethodDefinition*

1. Return **false**.

ClassElement : `static` *MethodDefinition*

1. Return **true**.

ClassElement : `;`

1. Return **false**.

Static Semantics: LexicallyDeclaredNames

ClassDeclaration: `class` *BindingIdentifier* *ClassTail*

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: PrototypeMethodDefinitions

ClassElementList : *ClassElement*

1. If *ClassElement* is the production *ClassElement* : `;` then, return a new empty List.
2. If IsStatic of *ClassElement* is **true**, return a new empty List.
3. If PropName of *ClassElement* is `"constructor"`, return a new empty List.
4. Return a List containing *ClassElement*.

ClassElementList : *ClassElementList* *ClassElement*

1. Let *list* be PrototypeMethodDefinitions of *ClassElementList*.
2. If *ClassElement* is the production *ClassElement* : `;` then, return *list*.
3. If IsStatic of *ClassElement* is **true**, return *list*.
4. If PropName of *ClassElement* is `"constructor"`, return *list*.
5.
6. Append *ClassElement* to the end of *list*.
7. Return *list*.

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*.

Deleted: true

Deleted: <#>If PropName of *ClassElement* is empty or `"constructor"`, return a new empty List.¶
<#>If IsStatic of *ClassElement* is **true**, return a new empt List.¶

Moved (insertion) [8]

Deleted: empty or

Moved up [8]: If IsStatic of *ClassElement* is **true**, retur *list*.¶

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*.

Static Semantics: PropName

ClassElement **:** **;**

1. Return empty.

Static Semantics: StaticPropertyNameList

ClassElementList **:** *ClassElement*

1. If PropName of *ClassElement* is empty, return a new empty List.
2. If IsStatic of *ClassElement* is **false**, return a new empty List.
3. Return a List containing PropName of *ClassElement*.

ClassElementList **:** *ClassElementList ClassElement*

1. Let *list* be StaticPropertyNameList of *ClassElementList*.
2. If PropName of *ClassElement* is empty, return *list*.
3. If IsStatic of *ClassElement* is **false**, return *list*.
4. Append PropName of *ClassElement* to the end of *list*.
5. Return *list*.

Static Semantics: StaticMethodDefinitions

ClassElementList **:** *ClassElement*

1. If *ClassElement* is the production *ClassElement* : **;** then, return a new empty List.
2. If IsStatic of *ClassElement* is **false**, return a new empty List.
3. Return a List containing *ClassElement*.

Deleted: <#>If PropName of *ClassElement* is empty, return a new empty List.¶

ClassElementList **:** *ClassElementList ClassElement*

1. Let *list* be StaticMethodDefinitions of *ClassElementList*.
2. If *ClassElement* is the production *ClassElement* : **;** then, return *list*.
3. If IsStatic of *ClassElement* is **false**, return *list*.
4. Append *ClassElement* to the end of *list*.
5. Return *list*.

Deleted: <#>If PropName of *ClassElement* is empty, return *list*.¶

Static Semantics: VarDeclaredNames

ClassDeclaration: `class` *BindingIdentifier ClassTail*

1. Return an empty List.

13.5.1.2 Runtime Semantics

Runtime Semantics: ClassDefinitionEvaluation

With parameter *className*.

ClassTail **:** *ClassHeritage*$_{opt}$ **{** *ClassBody* **}**

1. If *ClassHeritage*$_{opt}$ is not present, then
    a. let *protoParent* be the intrinsic object %ObjectPrototype%.
    b. Let *constructorParent* be the intrinsic object %FunctionPrototype%.
2. Else
    a. Let *superclass* be the result of evaluating *ClassHeritage*.
    b. ReturnIfAbrupt(*superclass*).
    c. If *superclass* is **null**, then
        i. Let *protoParent* be **null**.
        ii. Let *constructorParent* be the intrinsic object %FunctionPrototype%.
    d. Else if Type(*superclass*) is not Object, throw a **TypeError** exception.
    e. Else if IsConstructor(*superclass*) is **false**, then throw a **TypeError** exception.
    f. Else
        i. Let *protoParent* be the result of Get(*superclass*, `"prototype"`).
        ii. ReturnIfAbrupt(*protoParent*).
        iii. If Type(*protoParent*) is neither Object or Null, throw a **TypeError** exception.
        iv. Let *constructorParent* be *superclass*.
3. Let *proto* be the result of the abstract operation ObjectCreate with argument *protoParent*.
4. Let *lex* be the LexicalEnvironment of the running execution context.
5. If *className* is not **undefined**, then
    a. Let *scope* be the result of calling NewDeclarativeEnvironment passing *lex* as the argument
    b. Let *envRec* be *scope's* environment record.
    c. Call the CreateImmutableBinding concrete method of *envRec* passing *className* as the argument.
    d. Set the running execution context's LexicalEnvironment to *scope*.
6. Let *constructor* be ConstructorMethod of *ClassBody*.
7. If *constructor* is empty, then
    a. If *ClassHeritage*$_{opt}$ is present, then
        i. Let *constructor* be the result of parsing the String `"constructor(... args){super (...args);}"` using the syntactic grammar with the goal symbol *MethodDefinition*.
    b. Else,
        i. Let *constructor* be the result of parsing the String `"constructor( ){ }"` using the syntactic grammar with the goal symbol *MethodDefinition*.
8. Let *strict* be **true**.
9. Let *F* be the result of performing Property Definition Evaluation for *constructor* with argument *proto* and *constructorParent* as the optional *functionPrototype* argument.
10. Perform the abstract operation MakeConstructor with argument *F* and **false** as the optional *writablePrototype* argument and *proto* as the optional *prototype* argument.
11. If *className* is not **undefined**, then
    a. Call the InitialiseBinding concrete method of *envRec* passing *className* and *F* as the arguments.
12. Let *desc* be the Property Descriptor{[[Enumerable]]: **false**, [[Writable]]: **true**, [[Configurable]]: **true**}.
13. Call the [[DefineOwnProperty]] internal method of *proto* with arguments `"constructor"` and *desc*.
14. Let *protoMethods* be PrototypeMethodDefinitions of *ClassBody*.
15. For each *MethodDefinition m* in order from *protoMethods*
    a. Perform Property Definition Evaluation for *m* with argument *proto*.
16. Let *staticMethods* be StaticMethodDefinitions of *ClassBody*.
17. For each *MethodDefinition s* in order from *staticMethods*
    a. Perform Property Definition Evaluation for *s* with argument *F*.
18. Set the running execution context's LexicalEnvironment to *lex*.
19. Return *F*.

Runtime Semantics: Evaluation

ClassDeclaration: **`class`** *BindingIdentifier ClassTail*

1. Let *value* be the result of ClassDefinitionEvaluation of *ClassTail* with argument **undefined**.
2. ReturnIfAbrupt(*value*).
3. Let *env* be the running execution context's LexicalEnvironment.
4. Let *status* be the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.
5. ReturnIfAbrupt(*status*).

Deleted:
Deleted: does not have a [[Construct]] internal propertymethod
Deleted: ¶
Deleted: Let *protoParent* be *superclass*.
Deleted: <#>Let *constructorParent* be the intrinsi object %FunctionPrototype%.¶
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: passing
Deleted: as the argument
Deleted: *proto*
Deleted: <#>If the *ClassTail* is contained in strict code if its *FunctionBody* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.¶
Deleted: *ClassHeritage*$_{opt}$ is present and
Deleted: `.constructor`
Comment [AWB9130]: Note that this variable curre isn't used in this algorithm
Deleted: If the *ClassTail* is contained in strict code or if *constructor* is strict code, then l
Deleted: Otherwise let *strict* be **false**.
Comment [AWB8131]: As it now stands, this will ne be an abrupt completion
Deleted: ing
Deleted: l
Deleted: **false**
Deleted: **false**
Deleted: *object*
Deleted: ,
Deleted: , and **false**.
Deleted: *m*
Deleted: *m*
Deleted: <#>Let *className* be String Value of *BindingIdentifier*.¶
Deleted: className
Deleted: If *value* is an abrupt completion, return *value*

6. Return NormalCompletion(empty).

NOTE The argument to ClassDefinitionEvaluation controls whether or not the class that is defined with a BindingIdentifier has a local binding to the identifier. Only a *ClassExpression* gets a local name binding of its name. A *ClassDeclaration* never has such a binding. This maintains the parallel with *FunctionExpression* and *FunctionDeclaration*.

ClassExpression: **class** *BindingIdentifier*$_{opt}$ *ClassTail*

1. If *BindingIdentifier*$_{opt}$ is not present, then 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. ReturnIfAbrupt(*value*).
5. Return NormalCompletion(*value*).

13.6 Tail Position Calls

The wiki proposal has a preliminary attempt at defining tail position. See http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls.

This material still needs to be reviewed and updated for incorporation here.

13.6.1 Runtime Semantics

Runtime Semantics: PrepareForTailCall

The abstract operation PrepareForTailCall performs the following steps:

1. Let *leafContext* be the running execution context.
2. Suspend *leafContext*.
3. Pop *leafContext* from the execution context context stack. The execution context now on the top of the stack becomes the running execution context, however it remains in its suspended state.
4. Assert: *leafContext* has no further use. It will never be activated as the running execution context.

A tail position call must either release any transient internal resources associated with the currently executing function execution context before invoking the target function or reuse those resources in support of the target function.

NOTE 1 For example, a tail position call should only grow an implementation's activication 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.

14 Scripts and Modules

14.1 Script

Syntax

Script **:**
    ScriptBody$_{opt}$

ScriptBody **:**
    OuterStatementList

OuterStatementList **:**
    OuterItem
    OuterStatementList *OuterItem*

Deleted:

Deleted: Value

Comment [AWB9132]: TODO: Need to define tail positions.

Deleted: <#>13.1 Strict Mode Restrictions¶
<#>It is a SyntaxError if any *Identifier* value occurs more than once within a *FormalParameterList* of a strict mode *FunctionDeclaration* or *FunctionExpression*.
<#>It is a SyntaxError if the *Identifier* "eval" or the *Identifier* "arguments" occurs within a *FormalParameterList* of a strict mode *FunctionDeclaration* or *FunctionExpression*.¶
<#>It is a SyntaxError if the *Identifier* "eval" or the *Identifier* "arguments" occurs as the *Identifier* of a strict mode *FunctionDeclaration* or *FunctionExpression*.

Deleted: <#>13.21.156 . Creating Function Objects and Constructors¶
Runtime Semantics: FunctionCreate Abstract Operation¶
The abstract operation FunctionCreate requi the arguments: *kind* which is one of (Normal, Method, Arrow), A Function object is constructed as follows Given given an option parameter list specified by *FormalParameterL* a body specified by *FunctionBody*, a Lexical Environment specified by *Scope*, and a Boole flag *Strict*, and optionally, an object *functionPrototype*, an object *homeObject* and a string *methodName*. FunctionCreate performs following stepsa Function object is constructe as follows:¶
<#>Create a new native ECMAScript object an let *F* be that object.¶
<#>Set *F*'s essential internal methods except fo [[GetP]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.¶
<#>Set *F*'s essential internal methods for [[Ca [[GetP]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.19.¶
<#>Set all the internal methods, except for [[GetProperty]], of F as described in 8.12.¶
<#>Set Add the [[ClassIsFunctionNativeBrandBuiltinBrand]] internal data property with value NativeFunctionBuiltinFunction of to *F* to "Function".¶

Moved (insertion) [1]

Deleted: <#>13.2 Generator Definitio Syntax¶
GeneratorDeclaration :¶

Deleted: Programs

Deleted: *Program*

Deleted: *SourceElements*$_{opt}$

Deleted: *Program*

Deleted: *SourceElements*

Deleted: *Program*

Deleted: *SourceElement* .
SourceElements SourceElement

OuterItem **:**
ModuleDeclaration
ImportDeclaration
StatementListItem

14.1.1 Static Semantics

Static Semantics: Early Errors

ScriptBody **:** *OuterStatementList*

Deleted: *Program*

- It is a Syntax Error if the LexicallyDeclaredNames of *OuterStatementList* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *OuterStatementList* also occurs in the VarDeclaredNames of *OuterStatementList*.
- It is a Syntax Error if *OuterStatementList* Contains *ReturnStatement*.
- It is a Syntax Error if *OuterStatementList* Contains **super**.
- It is a Syntax Error if *OuterStatementList* Contains *YieldExpression*.

NOTE Additional error conditions relating to conflicting or duplicate declarations are checked during module linking prior to evaluation of a *Script*. If any such errors are detected the *Script* is not evaluated.

Static Semantics: IsStrict

ScriptBody **:** *OuterStatementList*

1. If this *ScriptBody* is contained in strict code or if *OuterStatementList* is strict code, then return **true**. Otherwise, return **false**.

Comment [AWB10133]: Need a better definition

Static Semantics: LexicallyDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be LexicallyDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *ModuleDeclaration*

1. Return the BoundNames of *ModuleDeclaration*.

Deleted: *Import*

OuterItem **:** *ImportDeclaration*

1. Return the BoundNames of *ImportDeclaration*.

OuterItem **:** *StatementListItem*

1. Return TopLevelLexicallyDeclaredNames of *StatementListItem*.

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

Static Semantics: LexicallyScopedDeclarations

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *declarations* be LexicallyScopedDeclarations of *OuterStatementList*.
2. Append to *declarations* the elements of the LexicallyScopedDeclarations of *OuterItem*.
3. Return *declarations*.

OuterItem **:** *ModuleDeclaration*

1. Return a new List containing *ModuleDeclaration*.

OuterItem **:** *ImportDeclaration*

1. Return a new List containing *ImportDeclaration*.

OuterItem **:** *StatementListItem*

1. Return TopLevelLexicallyScopedDeclarations of *StatementListItem*.

Static Semantics: VarDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be VarDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the VarDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *ModuleDeclaration*

2. Return an empty List.

OuterItem **:** *ImportDeclaration*

2. Return an empty List.

OuterItem **:** *StatementListItem*

2. Return TopLevelVarDeclaredNames of *StatementListItem*.

Static Semantics: VarScopedDeclarations

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *declarations* be VarScopedDeclarations of *OuterStatementList*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *OuterItem*.
3. Return *declarations*.

OuterItem **:** *ModuleDeclaration*

1. Return a new empty List.

OuterItem **:** *ImportDeclaration*

1. Return a new empty List.

OuterItem **:** *StatementListItem*

1. Return the TopLevelVarScopedDeclarations of *StatementListItem*.

14.1.2 Runtime Semantics

Runtime Semantics: Script Evaluation

With argument *realm* and *deletableBindings*.

Script **:** *ScriptBody*$_{opt}$

1. The code of this *Script* is strict mode code if the Directive Prologue (14.1) of its *ScriptBody* contains a Use Strict Directive or if any of the conditions of 10.1.1 apply. If the code of this *Script* is strict mode code,

Deleted: <#>SourceElement :¶
<#>Statement
FunctionDeclaration¶

Deleted: Program

Deleted: *Semantics*¶
The production

Deleted: *Program*

Deleted: *SourceElements*$_{opt}$

Deleted: *Program*

Deleted: is evaluated as follows:

Deleted: *Program*

Deleted: *SourceElements*

Deleted: *Program*

Deleted: *Program*

DRAFT

ScriptBody is evaluated in the following steps as strict mode code. Otherwise *ScriptBody* is evaluated in the following steps as non-strict mode code.

2. If *ScriptBody* is not present, return NormalCompletion(empty).
3. Let *globalEnv* be *realm*.[[globalEnv]].
4. Let *status* be the result of performing Global Declaration Instantiation as described in 10.5.1 using *ScriptBody*, *globalEnv*, and *deletableBindings* as arguments.
5. ReturnIfAbrupt(*status*).
6. Let *progCxt* be a new ECMAScript code execution context.
7. Set the *progCxt's* Realm to *realm*.
8. Set the *progCxt's* VariableEnvironment to *globalEnv*.
9. Set the *progCxt's* LexicalEnvironment to *globalEnv*.
10. If there is a currently running execution context, suspend it.
11. Push *progCxt* on to the execution context stack; *progCxt* is now the running execution context.
12. Let *result* be the result of evaluating *ScriptBody*.
13. Suspend *progCxt* and remove it from the execution context stack.
14. If the execution context stack is not empty, resume the context that is now on the top of the execution context stack as the running execution context. Otherwise, the execution context stack is now empty and there is no running execution context.
15. Return *result*.

NOTE The processes for initiating the evaluation of a *Script* and for dealing with the result of such an evaluation are defined by an ECMAScript implementation and not by this specification.

Runtime Semantics: Evaluation

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *sl* be the result of evaluating *OuterStatementList*.
2. ReturnIfAbrupt(*sl*).
3. Let *s* be the result of evaluating *OuterItem*.
4. If *s*.[[type]] is throw, return *s*.
5. If *s*.[[value]] is empty, let *V* = *sl*.[[value]], otherwise let *V* = *s*.[[value]].
6. Return Completion {[[type]]: *s*.[[type]], [[value]]: *V*, [[target]]: *s*.[[target]]}.

NOTE See the 12.1 NOTE regarding evaluation of *StatementList* **:** *StatementList StatementListItem*.

14.2 Directive Prologues and the Use Strict Directive

A Directive Prologue is the longest sequence of *ExpressionStatement* productions occurring as the initial *StatementListItem* productions of a *ScriptBody* or *FunctionBody* and where each *ExpressionStatement* in the sequence consists entirely of a *StringLiteral* token followed by a semicolon. The semicolon may appear explicitly or may be inserted by automatic semicolon insertion. A Directive Prologue may be an empty sequence.

A Use Strict Directive is an *ExpressionStatement* in a Directive Prologue whose *StringLiteral* is either the exact character sequences `"use strict"` or `'use strict'`. A Use Strict Directive may not contain an *EscapeSequence* or *LineContinuation*.

A Directive Prologue may contain more than one Use Strict Directive. However, an implementation may issue a warning if this occurs.

NOTE The *ExpressionStatement* productions of a Directive Prologue are evaluated normally during evaluation of the containing production. Implementations may define implementation specific meanings for *ExpressionStatement* productions which are not a Use Strict Directive and which occur in a Directive Prologue. If an appropriate notification mechanism exists, an implementation should issue a warning if it encounters in a Directive Prologue an *ExpressionStatement* that is not a Use Strict Directive or which does not have a meaning defined by the implementation.

Deleted: *SourceElements*
Deleted: *ProgramBody*
Deleted: *SourceElements*
Deleted: *Program*
Deleted: *SourceElements*
Deleted: *ProgramBody*
Deleted: (normal,
Deleted:
Deleted: , empty
Deleted: Binding
Deleted: for global code as described in 10.4.1
Deleted: <#>Set the *progCxt's* PreviousContext to emp
Deleted: *realm's* .[[Gglobal Environment]]
Deleted: *realm*.[[globalEnv]]
Deleted: 's Global Environment
Deleted: <#>If ProgramBody is present, then¶
<#>Perform Global Binding Instantiation as described in 10.5.x using the global code.¶
Deleted: *SourceElements*
Deleted: *ProgramBody*
Deleted:
Comment [AWB6134]: This is a place where we coi put something about what to do with unhandled exceptions
Deleted: Exit the execution context *progCxt*.
Deleted: T
Deleted: running current execution context to empty
Deleted: *Program*
Comment [AWB11135]: Not strictly true any longer. Will need to update.
Deleted: <#>The production *SourceElements* *ProgramBody* : *SourceElements* *SourceElementStatementList* is evaluated as follows
<#>Let headResult be the result of evaluating SourceElements.¶
<#>If headResult is an abrupt completion, retur headResult.¶
<#>Let tailResult be result of evaluating SourceElement.¶
<#>If tailResult.value is empty, let V = headResult.value, otherwise let V = tailResult.va
<#>Return (tailResult.type, V, tailResult.target)t result of evaluating StatementList. ¶
Deleted: <#>The production SourceElement : Statement is evaluated as follows:¶
<#>Return the result of evaluating Statement.¶
<#>The production SourceElement : FunctionDeclaration is evaluated as follows:¶
<#>Return (normal, empty, empty).¶
Deleted: as the initial *SourceElement* productions of a
Comment [AWB15136]: TODO: need to update
Deleted: *Program*
Deleted: *SourceElements*

14.3 Modules

15 Standard Built-in ECMAScript Objects

There are certain built-in objects available whenever an ECMAScript program 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.

Unless specified otherwise, a built-in object that is callable as a function is a Build-in Function object with the characteristics described in 8.4.7. Unless specified otherwise, the [[Extensible]] internal data property of a built-in object initially has the value **true**.

Many built-in objects are functions: they can be invoked with arguments. Some of them furthermore are constructors: they are functions intended for use with the **`new`** operator. For each built-in function, this specification describes the arguments required by that function and properties of the Function object. For each built-in constructor, this specification furthermore describes properties of the prototype object of that constructor and properties of specific object instances returned by a **`new`** expression that invokes that constructor.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this clause is given fewer arguments than the function is specified to require, the function or constructor shall behave exactly as if it had been given sufficient additional arguments, each such argument being the **undefined** value.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this clause is given more arguments than the function is specified to allow, the extra arguments are evaluated by the call and then ignored by the function. However, an implementation may define implementation specific behaviour relating to such arguments as long as the behaviour is not the throwing of a **TypeError** exception that is predicated simply on the presence of an extra argument.

NOTE Implementations that add additional capabilities to the set of built-in functions are encouraged to do so by adding new functions rather than adding new parameters to existing functions.

Every built-in function and every built-in constructor has the Function prototype object, which is the initial value of the expression **`Function.prototype`** (15.3.3), as the value of its [[Prototype]] internal data property.

Unless otherwise specified every built-in prototype object has the Object prototype object, which is the initial value of the expression **`Object.prototype`** (15.2.4), as the value of its [[Prototype]] internal data property, except the Object prototype object itself.

None of the built-in functions described in this clause that are not constructors shall implement the [[Construct]] internal method unless otherwise specified in the description of a particular function. The behaviour specified in this clause for each built-in function is the specification of the [[Call]] internal method behaviour for that function with the [[Call]] *thisArgument* providing the **this** value and the [[Call]] *argumentsList* providing the named parameters for each built-in function. 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. None of the built-in functions described in this clause shall have a **`prototype`** property unless otherwise specified in the description of a particular function.

This clause generally describes distinct behaviours for when a constructor is "called as a function" and for when it is "called as part of a **new** expression". The "called as a function" behaviour corresponds to the invocation of the constructor's [[Call]] internal method and the "called as part of a new expression" behaviour corresponds to the invocation of the constructor's [[Construct]] internal method.

Every built-in Function object, *F*, described in this clause—whether as a constructor, an ordinary function, or both—has the properties that are defined by performing the following step when the function object is created:

1. Perform the AddRestrictedFunctionProperties (13.6.3) abstract operation with argument *F*.

Deleted: the

Deleted: [[Class]] internal property of

Deleted: is **`"Function"`**

Deleted: has the [[IsFunctionNativeBrandBuiltinBrand]] interna data property with value NativeFunctionBuiltinFunction if that built-in object has a [[Call]] internal propertymethod

Deleted: , or **`"Object"`** if that built-in object does not have a [[Call]] internal property

Deleted: 15.3.4

Comment [AW137]: https://bugs.ecmascri org/show_bug.cgi?id=155

Deleted: .

Deleted: ¶

Deleted: operator

Every built-in Function object described in this clause—whether as a constructor, an ordinary function, or both—has a `length` property whose value is an integer. Unless otherwise specified, this value is equal to the largest number of named arguments shown in the subclause headings for the function description, including optional parameters.

NOTE For example, the Function object that is the initial value of the `slice` property of the String prototype object is described under the subclause heading "String.prototype.slice (start, end)" which shows the two named arguments start and end; therefore the value of the `length` property of that Function object is `2`.

In every case, the `length` property of a built-in Function object described in this clause has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Deleted: }.

Every other data property described in this clause has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** } unless otherwise specified.

Every accessor property described in this clause has the attributes {[[Enumerable]]: **false**, [[Configurable]]: **true** } unless otherwise specified. If only a get accessor function is described, the set accessor function is the default value, **undefined**. If only a set accessor is function is described the get accessor is the default value, **undefined**.

15.1 The Global Object

The unique *global object* is created before control enters any execution context.

Unless otherwise specified, the standard built-in properties of the global object have attributes {[[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.

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.

Deleted: property

The global object does not have a [[Call]] internal method; it is not possible to invoke the global object as a function.

Deleted: property

The value of the [[Prototype]] internal data property of the global object is implementation-dependent.

Deleted: s

Deleted: and [[Class]]

Deleted: ies

Deleted: are

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.

15.1.1 Value Properties of the Global Object

15.1.1.1 NaN

The value of `NaN` is **NaN** (see 8.5). This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.1.1.2 Infinity

The value of `Infinity` is **+∞** (see 8.5). This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.1.1.3 undefined

The value of `undefined` is **undefined** (see 8.1). This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.1.2 Function Properties of the Global Object

15.1.2.1 eval (x)

When the **eval** function is called with one argument *x*, the following steps are taken:

1. If Type(*x*) is not String, return *x*.
2. Let *script* be the ECMAScript code that is the result of parsing *x*, interpreted as UTF-16 encoded Unicode text as described in 8.4, for the goal symbol *Script*. If the parse fails or any early errors are detected, throw a **SyntaxError** exception (but see also clause 16).
3. If *script* Contains *ScriptBody* is **false**, return **undefined**.
4. Let *strictScript* be IsStrict of *script*.
5. If this is a direct call to eval (15.1.2.1.1), let *direct* be **true**, otherwise let *direct* be **false**.
6. If *direct* is **true** and the code that made the direct call to eval is strict code, then let *strictCaller* be **true**. Otherwise, let *strictCaller* be **false**.
7. Let *ctx* be the running execution context. If *direct* is **true** *ctx* will be the execution context that performed the direct eval. If *direct* is **false** *ctx* will be the execution context for the invocation of the eval function.
8. Let *evalRealm* be *ctx's* Realm.
9. If *direct* is **false** and *strictScript* is **false**, then
   a. Return the result of Script Evaluation for *script* with arguments *evalRealm* and **true**.
10. If *direct* is **true**, *strictScript* is **false**, *strictCaller* is **false**, and *ctx's* LexicalEnvironment is the same as *evalRealm*.[[globalEnv]], then
    a. Return the result of Script Evaluation for *script* with arguments *evalRealm* and **true**.
11. If *direct* is **true**, then
    a. If the code that made the direct call to eval is function code and ValidInFunction of *script* is **false**, then throw a **SyntaxError** exception.
    b. If the code that made the direct call to eval is module code and ValidInModule of *script* is **false**, then throw a **SyntaxError** exception.
12. If *direct* is **true**, then
    a. Let *lexEnv* be *ctx's* LexicalEnvironment.
    b. Let *varEnv* be *ctx's* VariableEnvironment.
13. Else,
    a. Let *lexEnv* be *evalRealm*.[[globalEnv]].
    b. Let *varEnv* be *evalRealm*.[[globalEnv]].
14. If *strictScript* is **true** or if *direct* is **true** and *strictCaller* is **true**, then
    a. Let *strictVarEnv* be the result of calling NewDeclarativeEnvironment passing *lexEnv* as the argument.
    b. Let *lexEnv* be *strictVarEnv*.
    c. Let *varEnv* be *strictVarEnv*.
15. Let *status* be the result of performing Eval Declaration Instantiation as described in 10.5.5 with *script*, *varEnv*, and *lexEnv*.
16. ReturnIfAbrupt(*status*).
17. Let *evalCxt* be a new ECMAScript code execution context.
18. Set the *evalCxt's* Realm to *evalRealm*.
19. Set the *evalCxt's* VariableEnvironment to *varEnv*.
20. Set the *evalCxt's* LexicalEnvironment to *lexEnv*.
21. If there is a currently running execution context, suspend it.
22. Push *evalCxt* on to the execution context stack; *evalCxt* is now the running execution context.
23. Let *result* be the result of evaluating *script*.
24. Suspend *evalCxt* and remove it from the execution context stack.
25. Resume the context that is now on the top of the execution context stack as the running execution context.
26. Return *result*.

NOTE The eval code cannot instantiate variable or function bindings in the variable environment of the calling context that invoked the eval if either the code of the calling context or the eval code is strict code. Instead such bindings are instantiated in a new VariableEnvironment that is only accessible to the eval code.

Deleted: *prog*
Deleted: as a
Deleted: *Program*
Deleted: k
Deleted: *staticScript*
Deleted: *staticScript*
Comment [AWB11138]: or eval code that was direct eval'ed by function code??
Comment [AWB11139]: TODO
Deleted: k
Comment [AWB11140]: or module code t was direct eval'ed by function code??
Comment [AWB11141]: TODO
Deleted: k
Deleted: <#>Let *evalCtx* be the result of establishing a new execution context (10.4.2) fo the eval code *prog*.¶
<#>If there is no calling context or if the eval c is not being evaluated by a direct call (15.1.2.1.1 to the eval function then,¶
<#>Initialise the execution context as if it wa global execution context using the eval code a *C* as described in 10.4.1.1.¶
Else…f *direct* is **true**, then,
Deleted: <#>Set the ThisBinding to the sa value as the ThisBinding of the calling execu context.¶
Set the…et *Lexical…exEnvironment*
Deleted: to
Deleted: the same value as the…*tx's*
Deleted: Set the…et *Variable*
Deleted: toironment
Deleted: to the same value as
Deleted: to
Deleted: to
Deleted: *staticScript*
Deleted: the eval code is strict code
Deleted: the
Deleted: LexicalEnvironment
Deleted: Set
Deleted: the …*exEnv* LexicalEnvironment
Deleted: to
Deleted: Set
Deleted: the …*arEnv* VariableEnvironment
Deleted: to
Deleted: ¶
Deleted: Perform …val Declaration Binding
Deleted: *realm*
Deleted: *the program …criptprog*
Deleted: <#>Exit the running execution conte *evalCtx*, restoring the previous execution contex

15.1.2.1.1 **Direct Call to Eval**

A direct call to the eval function is one that is expressed as a *CallExpression* that meets the following two conditions:

The Reference that is the result of evaluating the *MemberExpression* in the *CallExpression* has an environment record as its base value and its reference name is **`"eval"`**.

The result of calling the abstract operation GetValue with that Reference as the argument is the standard built-in function defined in 15.1.2.1.

15.1.2.2 **parseInt (string , radix)**

The **`parseInt`** function produces an integer value dictated by interpretation of the contents of the *string* argument according to the specified *radix*. Leading white space in *string* is ignored. If *radix* is **undefined** or 0, it is assumed to be 10 except when the number begins with the character pairs **`0x`** or **`0X`**, in which case a radix of 16 is assumed. If *radix* is 16, the number may also optionally begin with the character pairs **`0x`** or **`0X`**.

When the **`parseInt`** function is called, the following steps are taken:

1. Let *inputString* be ToString(*string*).
2. ReturnIfAbrupt(*string*).
3. Let *S* be a newly created substring of *inputString* consisting of the first character that is not a *StrWhiteSpaceChar* and all characters following that character. (In other words, remove leading white space.) If *inputString* does not contain any such characters, let *S* be the empty string.
4. Let *sign* be 1.
5. If *S* is not empty and the first character of *S* is a minus sign **-**, let *sign* be −1.
6. If *S* is not empty and the first character of *S* is a plus sign **+** or a minus sign **-**, then remove the first character from *S*.
7. Let *R* = ToInt32(*radix*).
8. ReturnIfAbrupt(*R*).
9. Let *stripPrefix* be **true**.
10. If *R* ≠ 0, then
    a. If *R* < 2 or *R* > 36, then return **NaN**.
    b. If *R* ≠ 16, let *stripPrefix* be **false**.
11. Else *R* = 0,
    a. Let *R* = 10.
12. If *stripPrefix* is **true**, then
    a. If the length of *S* is at least 2 and the first two characters of *S* are either “**0x**” or “**0X**”, then remove the first two characters from *S* and let *R* = 16.
13. If *S* contains any character that is not a radix-*R* digit, then let *Z* be the substring of *S* consisting of all characters before the first such character; otherwise, let *Z* be *S*.
14. If *Z* is empty, return **NaN**.
15. Let *mathInt* be the mathematical integer value that is represented by *Z* in radix-*R* notation, using the letters **A**-**Z** and **a**-**z** for digits with values 10 through 35. (However, if *R* is 10 and *Z* contains more than 20 significant digits, every significant digit after the 20th may be replaced by a **0** digit, at the option of the implementation; and if *R* is not 2, 4, 8, 10, 16, or 32, then *mathInt* may be an implementation-dependent approximation to the mathematical integer value that is represented by *Z* in radix-*R* notation.)
16. Let *number* be the Number value for *mathInt*.
17. Return *sign* × *number*.

Deleted: ,

NOTE **`parseInt`** may interpret only a leading portion of *string* as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.

15.1.2.3 **parseFloat (string)**

The **`parseFloat`** function produces a Number value dictated by interpretation of the contents of the *string* argument as a decimal literal.

When the **`parseFloat`** function is called, the following steps are taken:

1. Let *inputString* be ToString(*string*).
2. ReturnIfAbrupt(*string*).
3. Let *trimmedString* be a substring of *inputString* consisting of the leftmost character that is not a *StrWhiteSpaceChar* and all characters to the right of that character. (In other words, remove leading white space.) If *inputString* does not contain any such characters, let *trimmedString* be the empty string.
4. If neither *trimmedString* nor any prefix of *trimmedString* satisfies the syntax of a *StrDecimalLiteral* (see 9.3.1), return **NaN**.
5. Let *numberString* be the longest prefix of *trimmedString*, which might be *trimmedString* itself, that satisfies the syntax of a *StrDecimalLiteral*.
6. Return the Number value for the MV of *numberString*.

NOTE **`parseFloat`** may interpret only a leading portion of *string* as a Number value; it ignores any characters that cannot be interpreted as part of the notation of an decimal literal, and no indication is given that any such characters were ignored.

15.1.2.4 isNaN (number)

Returns **true** if the argument coerces to **NaN**, and otherwise returns **false**.

1. Let *num* be ToNumber(*number*).
2. ReturnIfAbrupt(*num*).
3. If *num* is **NaN**, return **true**.
4. Otherwise, return **false**.

Deleted: *ToNumber(number)*

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**.

15.1.2.5 isFinite (number)

Returns **false** if the argument coerces to **NaN**, **+∞**, or **–∞**, and otherwise returns **true**.

1. Let *num* be ToNumber(*number*).
2. ReturnIfAbrupt(*num*).
3. If ToNumber(*num*) is **NaN**, **+∞**, or **–∞**, return **false**.
4. Otherwise, return **true**.

Deleted: *ber*

15.1.3 URI Handling Function Properties

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 15.1.3.1, 15.1.3.2, 15.1.3.3 and 15.1.3.4.

NOTE Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

A URI is composed of a sequence of components separated by component separators. The general form is:

Scheme **:** *First* **/** *Second* **;** *Third* **?** *Fourth*

where the italicised names represent components and "**:**", "**/**", "**;**" and "**?**" are reserved characters used as separators. The **`encodeURI`** and **`decodeURI`** functions are intended to work with complete URIs; they assume that any reserved characters in the URI are intended to have special meaning and so are not encoded. The **`encodeURIComponent`** and **`decodeURIComponent`** functions are intended to work with the individual component parts of a URI; they assume that any reserved characters represent text and so must be encoded so that they are not interpreted as reserved characters when the component is part of a complete URI.

The following lexical grammar specifies the form of encoded URIs.

Syntax

uri **:::**
 uriCharacters$_{opt}$

uriCharacters **:::**
 uriCharacter uriCharacters$_{opt}$

uriCharacter **:::**
 uriReserved
 uriUnescaped
 uriEscaped

uriReserved **::: one of**
 `; / ? : @ & = + $ ,`

uriUnescaped **:::**
 uriAlpha
 DecimalDigit
 uriMark

uriEscaped **:::**
 `%` *HexDigit HexDigit*

uriAlpha **::: one of**
 `a b c d e f g h i j k l m n o p q r s t u v w x y z`
 `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

uriMark **::: one of**
 `- _ . ! ~ * ' ( )`

NOTE The above syntax is based upon RFC 2396 and does not reflect changes introduced by the more recent RFC 3986.

Runtime Semantics

When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is transformed into its UTF-8 encoding, with surrogate pairs first converted from UTF-16 to the corresponding code point value. (Note that for code units in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a String with each octet represented by an escape sequence of the form "`%xx`".

Runtime Semantics: Encode Abstract Operation

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 characters in *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat
    a. If *k* equals *strLen*, return *R*.
    b. Let *C* be the character at position *k* within *string*.
    c. If *C* is in *unescapedSet*, then
        i. Let *S* be a String containing only the character *C*.
        ii. Let *R* be 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.

Deleted: ,

ii. If the code unit value of *C* is less than 0xD800 or greater than 0xDBFF, then
    1. Let *V* be the code unit value of *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 character at position *k* within *string*.
    4. If *kChar* is less than 0xDC00 or greater than 0xDFFF, throw a **URIError** exception.
    5. Let *V* be (((the code unit value of *C*) – 0xD800) × 0x400 + (*kChar* – 0xDC00) + 0x10000).
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 position *j* within *Octets*.
    2. Let *S* be a String containing three characters "**%***XY*" where *XY* are two uppercase hexadecimal digits encoding the value of *jOctet*.
    3. Let *R* be a new String value computed by concatenating the previous value of *R* and *S*.
    4. Increase *j* by 1.

e. Increase *k* by 1.

Runtime Semantics: Decode Abstract Operation

The unescaping and decoding process is described by the abstract operation Decode taking two String arguments *string* and *reservedSet*.

1. Let *strLen* be the number of characters in *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat
    a. If *k* equals *strLen*, return *R*.
    b. Let *C* be the character at position *k* within *string*.
    c. If *C* is not '**%**', then
        i. Let *S* be the String containing only the character *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 characters at position (*k*+1) and (*k* + 2) within *string* do not represent hexadecimal digits, throw a **URIError** exception.
        iv. Let *B* be the 8-bit value represented by the two hexadecimal digits at position (*k* + 1) and (*k* + 2).
        v. Increment *k* by 2.
        vi. If the most significant bit in *B* is 0, then
            1. Let *C* be the character with code unit value *B*.
            2. If *C* is not in *reservedSet*, then
                a Let *S* be the String containing only the character *C*.
            3. Else *C* is in *reservedSet*.
                a Let *S* be the substring of *string* from position *start* to position *k* included.
        vii. Else the most significant bit in *B* is 1.
            1. Let *n* be the smallest non-negative number such that (*B* << *n*) & 0x80 is equal to 0.
            2. If *n* equals 1 or *n* is greater than 4, throw a **URIError** exception.
            3. Let *Octets* be an array of 8-bit integers of size *n*.
            4. Put *B* into *Octets* at position 0.
            5. If *k* + (3 × (*n* – 1)) is greater than or equal to *strLen*, throw a **URIError** exception.
            6. Let *j* be 1.
            7. Repeat, while *j* < *n*
                a Increment *k* by 1.
                b If the character at position *k* within *string* is not "%", throw a **URIError** exception.

Deleted: ,
Deleted: ,
Deleted: ,
Deleted: '
Deleted: '

c If the characters at position (*k* +1) and (*k* + 2) within *string* do not represent hexadecimal digits, throw a **URIError** exception.
d Let *B* be the 8-bit value represented by the two hexadecimal digits at position (*k* + 1) and (*k* + 2).
e If the two most significant bits in *B* are not 10, throw a **URIError** exception.
f Increment *k* by 2.
g Put *B* into *Octets* at position *j*.
h Increment *j* by 1.

8. Let *V* be the value obtained by applying the UTF-8 transformation to *Octets*, that is, from an array of octets into a 21-bit value. If *Octets* does not contain a valid UTF-8 encoding of a Unicode code point throw an **URIError** exception.
9. If V < 0x10000, then
   a Let *C* be the character with code unit value *V*.
   b If *C* is not in *reservedSet*, then
      i. Let *S* be the String containing only the character *C*.
   c Else *C* is in *reservedSet*,
      i. Let *S* be the substring of *string* from position *start* to position *k* included.
10. Else $V \geq$ 0x10000,
    a Let *L* be (((*V* – 0x10000) & 0x3FF) + 0xDC00).
    b Let *H* be ((((*V* – 0x10000) >> 10) & 0x3FF) + 0xD800).
    c Let *S* be the String containing the two characters with code unit values *H* and *L*.

e. Let *R* be a new String value computed by concatenating the previous value of *R* and *S*.
f. Increase *k* by 1.

NOTE This syntax of Uniform Resource Identifiers is based upon RFC 2396 and does not reflect the more recent RFC 3986 which replaces RFC 2396. A formal description and implementation of UTF-8 is given in RFC 3629.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of n octets, n>1, the initial octet has the n higher-order bits set to 1, followed by a bit set to 0. The remaining bits of that octet contain bits from the value of the character to be encoded. The following octets all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each to contain bits from the character to be encoded. The possible UTF-8 encodings of ECMAScript characters are specified in Table 32.

Table 32 — UTF-8 Encodings

Code Unit Value	*Representation*	*1st Octet*	*2nd Octet*	*3rd Octet*	*4th Octet*
0x0000 - 0x007F	00000000 0*zzzzzzz*	0*zzzzzzz*			
0x0080 - 0x07FF	00000*yyy yyzzzzzz*	110*yyyyy*	10*zzzzzz*		
0x0800 - 0xD7FF	*xxxxyyyy yyzzzzzz*	1110*xxxx*	10*yyyyyy*	10*zzzzzz*	
0xD800 - 0xDBFF *followed by* **0xDC00 – 0xDFFF**	110110*vv vvwwwwxx* *followed by* 110111*yy yyzzzzzz*	11110*uuu*	10*uuwwww*	10*xxyyyy*	10*zzzzzz*
0xD800 - 0xDBFF *not followed by* **0xDC00 – 0xDFFF**	*causes* **URIError**				
0xDC00 – 0xDFFF	*causes* **URIError**				
0xE000 - 0xFFFF	*xxxxyyyy yyzzzzzz*	1110*xxxx*	10*yyyyyy*	10*zzzzzz*	

Where

uuuuu = *vvvv* + 1

to account for the addition of 0x10000 as in Surrogates, section 3.7, of the Unicode Standard.

The range of code unit values 0xD800-0xDFFF is used to encode surrogate pairs; the above transformation combines a UTF-16 surrogate pair into a UTF-32 representation and encodes the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

Deleted: is less than

Deleted: ,

Deleted: ,

Deleted: is

RFC 3629 prohibits the decoding of invalid UTF-8 octet sequences. For example, the invalid sequence C0 80 must not decode into the character U+0000. Implementations of the Decode algorithm are required to throw a **URIError** when encountering such invalid sequences.

15.1.3.1 decodeURI (encodedURI)

The **`decodeURI`** function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the **`encodeURI`** function is replaced with the character that it represents. Escape sequences that could not have been introduced by **`encodeURI`** are not replaced.

When the **`decodeURI`** function is called with one argument *encodedURI*, the following steps are taken:

1. Let *uriString* be ToString(*encodedURI*).
2. ReturnIfAbrupt(*uriString*).
3. Let *reservedURISet* be a String containing one instance of each character valid in *uriReserved* plus "**#**".
4. Return the result of calling Decode(*uriString*, *reservedURISet*)

NOTE The character "**#**" is not decoded from escape sequences even though it is not a reserved URI character.

15.1.3.2 (encodedURIComponent)

The **`decodeURIComponent`** function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the **`encodeURIComponent`** function is replaced with the character that it represents.

When the **`decodeURIComponent`** function is called with one argument *encodedURIComponent*, the following steps are taken:

1. Let *componentString* be ToString(*encodedURIComponent*).
2. ReturnIfAbrupt(*componentString*).
3. Let *reservedURIComponentSet* be the empty String.
4. Return the result of calling Decode(*componentString*, *reservedURIComponentSet*)

15.1.3.3 encodeURI (uri)

The **`encodeURI`** function computes a new version of a URI in which each instance of certain characters is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

When the **`encodeURI`** function is called with one argument *uri*, the following steps are taken:

1. Let *uriString* be ToString(*uri*).
2. ReturnIfAbrupt(*uriString*).
3. Let *unescapedURISet* be a String containing one instance of each character valid in *uriReserved* and *uriUnescaped* plus "**#**".
4. Return the result of calling Encode(*uriString*, *unescapedURISet*)

NOTE The character "**#**" is not encoded to an escape sequence even though it is not a reserved or unescaped URI character.

15.1.3.4 encodeURIComponent (uriComponent)

The **`encodeURIComponent`** function computes a new version of a URI in which each instance of certain characters is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

When the **`encodeURIComponent`** function is called with one argument *uriComponent*, the following steps are taken:

1. Let *componentString* be ToString(*uriComponent*).
2. ReturnIfAbrupt(*componentString*).
3. Let *unescapedURIComponentSet* be a String containing one instance of each character valid in *uriUnescaped*.
4. Return the result of calling Encode(*componentString*, *unescapedURIComponentSet*)

15.1.4 Constructor Properties of the Global Object

15.1.4.1 **Object (. . .)**

See 15.2.1 and 15.2.2.

15.1.4.2 **Function (. . .)**

See 15.3.1.

Deleted: and 15.3.2

15.1.4.3 **Array (. . .)**

See 15.4.1.

Deleted: and 15.4.2

15.1.4.4 **String (. . .)**

See 15.5.1.

Deleted: and 15.5.2

15.1.4.5 **Boolean (. . .)**

See 15.6.1.

Deleted: and 15.6.2

15.1.4.6 **Number (. . .)**

See 15.7.1.

Deleted: and 15.7.2

15.1.4.7 **Date (. . .)**

See 15.9.2.

15.1.4.8 **RegExp (. . .)**

See 15.10.3.

Deleted: and 15.10.4

15.1.4.9 **Error (. . .)**

See 15.11.1 and 15.11.2.

15.1.4.10 **EvalError (. . .)**

See 15.11.6.1.

15.1.4.11 **RangeError (. . .)**

See 15.11.6.2.

15.1.4.12 **ReferenceError (. . .)**

See 15.11.6.3.

15.1.4.13 **SyntaxError (. . .)**

See 15.11.6.4.

15.1.4.14 TypeError (. . .)

See 15.11.6.5.

15.1.4.15 15.1.4.15 URIError (. . .)

See 15.11.6.6.

15.1.4.16 Map (. . .)

See 15.14.3.

15.1.4.17 WeakMap (. . .)

See 15.15.3.

15.1.4.18 15. Set (. . .)

See 15.16.3.

15.1.5 Other Properties of the Global Object

15.1.5.1 Math

See 15.8.

15.1.5.2 JSON

See 15.12.

15.2 Object Objects

15.2.1 The Object Constructor Called as a Function

When `Object` is called as a function rather than as a constructor, it performs a type conversion.

15.2.1.1 Object ([value])

When the `Object` function is called with no arguments or with one argument *value*, the following steps are taken:

1. If *value* is **null**, **undefined** or not supplied, return the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
2. Return ToObject(*value*).

15.2.2 The Object Constructor

When `Object` is called as part of a `new` expression, it is a constructor that may create an object.

15.2.2.1 new Object ([value])

When the `Object` constructor is called with no arguments or with one argument *value*, the following steps are taken:

1. If *value* is supplied, then
   a. If Type(*value*) is Object, then return *value*.
   b. If Type(*value*) is String, return ToObject(*value*).
   c. If Type(*value*) is Boolean, return ToObject(*value*).

Deleted: Runtime Semantics: ObjectCrea
Abstract Operation¶
The abstract operation ObjectCreate with optional argument *proto* (an object or null) is used to specify the creation of new Object objects. It performs the following steps:¶
<#>If *proto* was not provided, let *proto* be the intrinsic %ObjectPrototype%¶
<#>Let *obj* be a newly created ECMAScript object.¶
<#>Set the [[Prototype]] internal property of *ol* to *proto*.¶
<#>Set *obj*'s common internal methods to the default definitions specified in 8.12.¶
<#>Set the [[Extensible]] internal property of *o* to **true**.¶
<#>Return *obj*.¶
<#>¶

Deleted: create and

Deleted: ion

Deleted: ObjectCreate

Deleted: a new Object object exactly as if the standard built-in Object constructor had been ca with the same arguments (15.2.2.1)

Deleted: ¶
If the *value* is a native ECMAScript object, d not create a new object but simply

Deleted: <#>If the *value* is a host objec then actions are taken and a result is return in an implementation-dependent manner th may depend on the host object.¶

d. If Type(*value*) is Number, return ToObject(*value*).
2. Assert: The argument *value* was not supplied or its type was Null or Undefined.
3. Return the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.

15.2.3 Properties of the Object Constructor

The value of the [[Prototype]] internal data property of the Object constructor is the standard built-in Function prototype object.

Besides the **`length`** property (whose value is **1**), the Object constructor has the following properties:

15.2.3.1 Object.prototype

The initial value of **`Object.prototype`** is the standard built-in Object prototype object (15.2.4).

This property has the attributes {[[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.2.3.2 Object.getPrototypeOf (O)

When the **`getPrototypeOf`** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[GetInheritance]] internal method of *obj*.

15.2.3.3 Object.getOwnPropertyDescriptor (O, P)

When the **`getOwnPropertyDescriptor`** function is called, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*P*).
4. ReturnIfAbrupt(*key*).
5. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *key*.
6. ReturnIfAbrupt(*desc*).
7. Return the result of calling FromPropertyDescriptor(*desc*) (8.2.5.4).

15.2.3.4 Object.getOwnPropertyNames (O)

When the **getOwnPropertyNames** function is called, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.
4. ReturnIfAbrupt(*keys*).
5. Let *nameList* be a new empty List.
6. Let *gotAllNames* be **false**.
7. Repeat while *gotAllNames* is **false**,
    a. Let *next* be the result of Invoke(*keys*, **`"next"`**).
    b. ReturnIfAbrupt(*next*).
    c. If Type(*next*) is not Object, then throw a **TypeError** exception.
    d. Let *done* be IteratorComplete(*next*).
    e. ReturnIfAbrupt(*done*).
    f. If *done* is **true**, then let *gotAllNames* be **true**.
    g. Else,
        i. Let *nextKey* be IteratorValue(*next*).
        ii. ReturnIfAbrupt(*nextKey*).
        iii. If Type(*nextKey*) is String, then
            1. Append *nextKey* as the last element of *nameList*.

Deleted: <#>Let *obj* be a newly created native ECMAScript object.¶
<#>Set the [[Prototype]] internal property of *obj* to the standard built-in Object prototype object (15.2.4).¶
<#>Set the [[Class]] internal property of *obj* to **"Object**
<#>Set the [[Extensible]] internal property of *obj* to **true**.
<#>Set all the internal methods of *obj* as specified in 8.12

Deleted: obj

Deleted: ion

Deleted: the internal properties and

Deleted: <#>If Type(*O*) is not Object throw a **TypeErr** exception.¶

Deleted: value

Deleted: the

Deleted: Prototype

Deleted: e

Deleted: property

Deleted: *O*

Deleted: <#>If Type(*O*) is not Object throw a **TypeErr** exception.¶

Deleted: *name*

Deleted: ToString

Deleted: *name*

Deleted: *O*

Deleted: *name*

Deleted: 10

Deleted: <#>If Type(*O*) is not Object, throw a **TypeErr** exception.¶

Deleted: *O*

Deleted: IteratorComplete(*next*)

Deleted: <#>If *next* is an abrupt completion, then return *next*.¶
<#>Assert *next*.[[type]] is normal.¶
<#>Let *nextKey* be *next*.[[value]].¶

8. Return CreateArrayFromList(*keys*).

15.2.3.5 Object.create (O [, Properties])

The **create** function creates a new object with a specified prototype. When the **create** function is called, the following steps are taken:

1. If Type(*O*) is not Object or Null throw a **TypeError** exception.
2. Let *obj* be the result of the abstract operation ObjectCreate with argument *O*.
3. If the argument *Properties* is present and not **undefined**, then
   a. Return the result of the abstract operation ObjectDefineProperties with arguments *obj* and *Properties*.
4. Return *obj*.

15.2.3.6 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 *name* be ToPropertyKey(*P*).
3. ReturnIfAbrupt(*name*).
4. Let *desc* be the result of calling ToPropertyDescriptor with *Attributes* as the argument.
5. ReturnIfAbrupt(*desc*).
6. Let *success* be the result of DefinePropertyOrThrow(*O*, *name*, *desc*).
7. ReturnIfAbrupt(*success*).
8. Return *O*.

15.2.3.7 Object.defineProperties (O, Properties)

The **defineProperties** function is used to add own properties and/or update the attributes of existing own properties of an object. When the **defineProperties** function is called, the following steps are taken:

1. Return the result of the abstract operation ObjectDefineProperties with arguments *O* and *Properties*.

Runtime Semantics: ObjectDefineProperties Abstract Operation

The abstract operation ObjectDefineProperties with arguments *O* and *Properties* performs the following steps:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *props* be ToObject(*Properties*).
3. Let *names* be an internal list containing the keys of each enumerable own property of *props*.
4. Let *descriptors* be an empty internal List.
5. For each element *P* of *names* in list order,
   a. Let *descObj* be the result of Get(*props*, *P*).
   b. ReturnIfAbrupt(*descObj*).
   c. Let *desc* be the result of calling ToPropertyDescriptor with *descObj* as the argument.
   d. ReturnIfAbrupt(*desc*).
   e. Append the pair (a two element List) consisting of *P* and *desc* to the end of *descriptors*.
6. Let *pendingException* be **undefined**.
7. For each *pair* from *descriptors* in list order,
   a. Let *P* be the first element of *pair*.
   b. Let *desc* be the second element of *pair*.
   c. Let *status* be the result of DefinePropertyOrThrow(*O*, *P*, *desc*).
   d. If *status* is an Abrupt Completion then,
      i. If *pendingException* is **undefined**, then set *pendingException* to *status*.
8. ReturnIfAbrupt(*pendingException*).
9. Return *O*.

If an implementation defines a specific order of enumeration for the for-in statement, that same enumeration order must be used to order the list elements in step 3 of this algorithm.

Deleted: <#>If Type(*O*) is not Object throw a **TypeError** exception.¶
<#>Let *keys* be the result of calling the [[OwnPropertyKeys]] internl method of *O*.¶
<#>ReturnIfAbrupt(*keys*).¶
<#>Let *array* be the result of the abstraction operationabstract operation ArrayCreate (15.4) with argument 0the result of creating a new object as if by the expression `new Array ()` where `Arr` is the standard built-in constructor with that nam
<#>Let *n* be 0.¶
<#>For each named own property *P* of *O*¶
<#>Let *name* be the String value that is the name of *P*.¶
<#>Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*n*), the PropertyDescriptor {[[Value]]: *name*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.¶
<#>Increment *n* by 1.¶
<#>Return *array*CreateArrayFromList(*keys*);.¶

Deleted: NOTE If *O* is a String instance, th set of own properties processed in step 4 includes the implicit properties defined in

Deleted: abstraction operation

Deleted: the result of creating a new object as

Deleted: <#>Set the [[Prototype]] internal

Deleted: add own properties to *obj* as if by

Deleted: `Object.defineProperties`

Deleted: ToString

Deleted: Call

Deleted: the [[

Deleted: Own

Deleted:]]

Deleted: internal method of

Deleted: with arguments

Deleted: , and **true**

Deleted: em

Deleted: s

Deleted: names

Deleted: calling the [[

Deleted:]] internal method of

Deleted: with

Deleted: as the argument

Deleted: Call

Deleted: calling the [[

Deleted: Own

Deleted:]]

Deleted: internal method of

Deleted: with arguments

Deleted: , and **true**

Deleted: **undefined**

Deleted: *status*

NOTE An exception in defining an individual property in step 7 does not terminate the process of defining other properties. All valid property definitions are processed.

15.2.3.8 Object.seal (O)

When the **seal** function is called, the following steps are taken:

1. If Type(*O*) is not Object, return *O*.
2. Let *status* be the result of SetIntegrityLevel(*O*, "**sealed**").
3. ReturnIfAbrupt(*status*).
4. If *status* is **false**, throw a **TypeError** exception.
5. Return *O*.

15.2.3.9 Object.freeze (O)

When the **freeze** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw, return *O*.
2. Let *status* be the result of SetIntegrityLevel(*O*, "**frozen**").
3. ReturnIfAbrupt(*status*).
4. If *status* is **false**, throw a **TypeError** exception.
5. Return *O*.

15.2.3.10 Object.preventExtensions (O)

When the **preventExtensions** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw, return *O*.
2. Let *status* be the result of calling the [[PreventExtensions]] internal method of ,
3. ReturnIfAbrupt(*status*).
4. If *status* is **false**, throw a **TypeError** exception.
5. Return *O*.

15.2.3.11 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**").

15.2.3.12 Object.isFrozen (O)

When the **isFrozen** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object, return **true**.
2. Return TestIntegrityLevel(*O*, "**frozen**").

15.2.3.13 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 the result of IsExtensible(*O*) .

15.2.3.14 Object.keys (O)

When the **keys** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.

Formatted: Note
Deleted: throw a **TypeError** exception
Deleted: <#>For each named own property name *P* of
Deleted: of
Deleted: MakeObjectSecure(
Deleted: *O*, with argument
Deleted: **P, desc**
Deleted: **false**)
Deleted: calling the [[DefineOwnProperty]] internal
Deleted: <#>Set the [[Extensible]] internal property of
Deleted: a **TypeError** exception
Deleted: <#>For each named own property name *P* of
Deleted:
Deleted: calling Call the [[DefineOwnProperty]] internal
Deleted: calling the [[SetIntegrity]] internal methods of
Deleted: with
Deleted: **true***P*, *desc*, and **true** as arguments
Deleted: <#>Set the [[Extensible]] internal property of
Deleted: a **TypeError** exception
Deleted: Set
Deleted: SetIntegrity
Deleted: Extensible
Deleted: property
Deleted: *O* with argument "**nonextensible**"
Deleted: to **false**
Deleted: throw a **TypeError** exception
Deleted: <#>For each named own property name *P* of
Deleted: the result of calling the [[HasIntegrity]] internal
Deleted: TestIfSecureObject(*O*, **false**)
Deleted: **false**
Deleted: throw a **TypeError** exception
Deleted: <#>For each named own property name *P* of
Deleted: the result of calling the [[HasIntegrity]] internal
Deleted: IfSecureObject
Deleted: **falsetrue**
Deleted: throw a **TypeError** exception
Deleted: calling the Boolean value of the [[
Deleted: Get
Deleted:]]
Deleted: internal property method of
Deleted: <#>If the Type(*O*) is not Object, throw a
Formatted
Deleted: Keys
Deleted: *O*

4. ReturnIfAbrupt(*keys*).
5. Let *nameList* be a new empty List.
6. Let *gotAllNames* be **false**.
7. Repeat while *gotAllNames* is **false**,
    a. Let *next* be the result of Invoke(*keys*, `"next"`).
    b. ReturnIfAbrupt(*next*).
    c. If Type(*next*) is not Object, then throw a **TypeError** exception.
    d. Let *done* be IteratorComplete(*next*).
    e. ReturnIfAbrupt(*done*).
    f. If *done* is **true**, then let *gotAllNames* be **true**.
    g. Else,
        i. Let *nextKey* be IteratorValue(*next*).
        ii. ReturnIfAbrupt(*nextKey*).
        iii. If Type(*nextKey*) is String, then
            1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *nextKey*.
            2. ReturnIfAbrupt(*desc*).
            3. If *desc* is not **undefined** and *desc*.[[Enumerable]] is **true**, then
                a. Append *nextKey* as the last element of *nameList*.
8. Return CreateArrayFromList(*keys*).

If an implementation defines a specific order of enumeration for the for-in statement, the same order must be used for the elements of the array returned in step 7.

15.2.3.15 Object.getOwnPropertyKeys (O)

When the **getOwnPropertyKeys** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.
4. Return *keys*.

TODO :

- May need to say something about order of enumeration and post call property additions and deletions

15.2.3.16 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*).

15.2.3.17 Object.assign (target, source)

TODO :

- Only enumerable own properties of source
- Invoke [[Get]] on property list derived from source, for each property in list [[Put]] on target
- private names are not copied
- unique names are copied
- super mechanism (rebind super)
- Returns modified "target"

Formatted: Outline numbered + Level: 2 + Numbering Style: a, b, c, … + Start at: 1 + Alignment: Left + Aligned at: 0.5" + Tab after: 0.75" + Indent at: 0.75"

Deleted: IteratorComplete(*next*)

Formatted: Outline numbered + Level: 3 + Numbering Style: i, ii, iii, … + Start at: 1 + Alignment: Right + Aligned at: 1" + Tab after: 1.25" + Indent at: 1.25"

Deleted: <#>If *next* is an abrupt completion, then return *next*.¶
<#>Assert *next*.[[type]] is normal.¶
<#>Let *nextKey* be *next*.[[value]].¶

Formatted: Outline numbered + Level: 4 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.5" + Tab after: 1.75" + Indent at: 1.75"

Formatted: Outline numbered + Level: 5 + Numbering Style: a, b, c, … + Start at: 1 + Alignment: Left + Aligned at: 2" + Tab after: 2.25" + Indent at: 2.25"

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

Deleted: <#>If Type(*O*) is not Object, throw **TypeError** exception.¶

Deleted: *O*

Deleted: \

Deleted: <#>Let *n* be the number of own enumerable properties of *O*¶
<#>Let *array* be the result of the abstraction operationabstract operation ArrayCreate (15.4) w argument *n*the result of creating a new Object as by the expression **`new Array(n)`** where **Arr** is the standard built-in constructor with that nam
<#>Let *index* be 0.¶
<#>For each own enumerable property of *O* whose name String is *P*¶
<#>Let *P* be the key of the property.¶
<#>Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*index*), the PropertyDescriptor {[[Value]]: *P*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**} and **false**.¶
<#>Increment *index* by 1.¶
<#>Return *array*.¶
If an implementation defines a specific order enumeration for the for-in statement, that san enumeration order must be used in step 5 of this algorithm.¶

Comment [AWB11142]: TODO need an algorithm

Because [[Get]]/[[Put]] are invoked we need t figure out something to say about the order in which properties are accessed.

15.2.3.18 Object.mixin (target, source)

Don't have TC39 concensus on including this or its name. Hower, there appears to be strong interest both within TC39 and on es-disuss and reasonable use cases. "mixin" seems to be the favorate name from es-discuss, although concerns have been raised that it might clash with some existing libraries.

TODO :

- All non-private properties of source
- Uses [[GetOwnProperty]]/[[DefineOwnProperty]]
- private symbols are not copied
- non-symbols are copied
- super mechanism (rebind super)
- Returns modified "target"

15.2.3.19 Object.setPrototypeOf (O, proto)

When the `setPrototypeOf` function is called with arguments *O* and proto, the following steps are taken:

1. If Type(*O*) is not Object, then throw a **TypeError** exception.
2. If Type(*proto*) is neither Object or Null, then throw a **TypeError** exception.
3. Let *status* be the result of calling the [[SetInheritance]] internal method of *O* with argument *proto*.
4. ReturnIfAbrupt(*status*).
5. If *status* is **false**, then throw a **TypeError** exception.
6. Return *O*.

15.2.4 Properties of the Object Prototype Object

The Object prototype object is an ordinary object.

The value of the [[Prototype]] internal data property of the Object prototype object is **null** and the initial value of the [[Extensible]] internal data property is **true**.

15.2.4.1 Object.prototype.constructor

The initial value of `Object.prototype.constructor` is the standard built-in `Object` constructor.

15.2.4.2 Object.prototype.toString ( )

When the `toString` method is called, the following steps are taken:

1. If the **this** value is **undefined**, return `"[object Undefined]"`.
2. If the **this** value is **null**, return `"[object Null]"`.
3. Let *O* be the result of calling ToObject passing the **this** value as the argument.
4. If *O* is an exotic Symbol object, then return `"[object Symbol]"`.
5. Else, if *O* is an exotic Array object, then let *builtinTag* be `"Array"`.
6. Else, if *O* is an exotic String object, then let *builtinTag* be `"String"`.
7. Else, if *O* is an exotic arguments object, then let *builtinTag* be `"Arguments"`.

Deleted: ¶

Deleted: <#>15.2.3.15 Object.isObject (O)¶
<#>When the isObject function is called with argument *O*, the following steps are taken:¶
<#>If Type(*O*) is Object return true.¶
<#>Return false.¶

Deleted: , the value of the [[Class]] internal property is `"Object"`,

Deleted: ¶

Moved down [2]: If Type(the **this** value) is Symbol, th return `"[object Symbol]"`.¶

Moved (insertion) [2]

Deleted: Type(the **this** value) is

Deleted: <#>If *O* is an exotic Symbol object, then retur `"[object` let *tag* be `"Symbol]"`.¶

Deleted: I

8. Else, if *O* is an ordinary function object, a built-in function object, or a bound function exotic object, then let *builtinTag* be `"Function"`.
9. Else, if *O* has a [[ErrorData]] internal data property, then let *builtinTag* be `"Error"`.
10. Else, if *O* has a [[BooleanData]] internal data property, then let *builtinTag* be `"Boolean"`.
11. Else, if *O* has a [[NumberData]] internal data property, then let *builtinTag* be `"Number"`.
12. Else, if *O* has a [[DateValue]] internal data property, then let *builtinTag* be `"Date"`.
13. Else, if *O* has a [[RegExpMatcher]] internal data property, then let *builtinTag* be `"RegExp"`.
14. Else, if *O* has a [[MathTag]] internal data property, then let *builtinTag* be `"Math"`.
15. Else, if *O* has a [[JSONTag]] internal data property, then let *builtinTag* be `"JSON"`.
16. Else, let *builtinTag* be `"Object"`.
17. Let *hasTag* be the result of HasProperty(*O*, @@toStringTag).
18. ReturnIfAbrupt(*hasTag*).
19. If *hasTag* is **false**, then let *tag* be *builtinTag*.
20. Else,
    a. Let *tag* be the result of Get(*O*, @@toStringTag).
    b. If *tag* is an abrupt completion, let *tag* be NormalCompletion(`"???"`).
    c. Let *tag* be *tag*.[[value]].
    d. If Type(*tag*) is not String, let *tag* be `"???"`.
    e. If *tag* is any of `"Arguments"`, `"Array"`, `"Boolean"`, `"Date"`, `"Error"`, `"Function"`, `"JSON"`, `"Math"`, `"Number"`, `"RegExp"`, or `"String"` and SameValue(*tag*, *builtinTag*) is **false**, then let *tag* be the string value `"~"` concatenated with the current value of *tag*.
21. Return the String value that is the result of concatenating the three Strings "[object ", *tag*, and "]".

NOTE Historically, this function was occasionally used to access the string value of the [[Class]] internal data property that was used in previous editions of this specification as a nominal type tag for various built-in objects. The above definition of `toString` preserves the ability to use it as a reliable test for those specific kinds of built-in objects but it does not provide a reliable type testing mechanism for other kinds of built-in or program defined objects.

15.2.4.3 Object.prototype.toLocaleString ( )

When the **toLocaleString** method is called, the following steps are taken:

1. Let *O* be the **this** value.
2. ReturnIfAbrupt(*O*).
3. Return the result of Invoke(*O*, `"toString"`).

NOTE 1 This function is provided to give all Objects a generic `toLocaleString` interface, even though not all may use it. Currently, `Array`, `Number`, and `Date` provide their own locale-sensitive `toLocaleString` methods.

NOTE 2 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.2.4.4 Object.prototype.valueOf ( )

When the **valueOf** method is called, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. Return *O*.

15.2.4.5 Object.prototype.hasOwnProperty (V)

When the `hasOwnProperty` method is called with argument *V*, the following steps are taken:

1. Let *P* be ToPropertyKey(*V*).
2. ReturnIfAbrupt(*P*).
3. Let *O* be the result of calling ToObject passing the **this** value as the argument.
4. ReturnIfAbrupt(*O*).

Comment [AWB14143]: Need to decide h to tag the Math object

Comment [AWB14144]: Need to decide h to tag the JSON object

Deleted: <#>IElse if *O* has a [[NativeBrandBuiltinBrand]] internal data prope let *tag* be the corresponding value from Table 32Table 30Table 28.¶ <#>Else ¶

Deleted: calling the [[HasProperty]] internal method of

Deleted: with argument

Deleted: `"Object"`

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: i

Deleted: *O*

Deleted: a host object, let *tag* be an implementation defined string value. This val may not be

Deleted: `"Object"`,

Deleted: As

Deleted: .

Deleted: <#>Else, Let let class tag be the stri value "Object"of the [[Class]] internal property O.¶

Deleted: *class*

Deleted: ¶

Deleted: Table 323028 — [[BuiltinBrand]] Tag Values¶ [[NativeBrandBuiltinBrand]] Value

Deleted: This

Deleted: result of calling ToObject passing the

Deleted: as the argument

Deleted: <#>Let *toString* be the result of call the [[Get]] internal method of *O* passing **"toString"** as the argument.¶

Deleted: calling

Deleted: the abstract operation

Deleted: the

Deleted: with arguments

Deleted: `[[Call]] internal method`

Deleted: ,*toString* passing *O*,

Deleted: as the **this** value

Deleted: and

Deleted: no

Deleted: an empty arguments List

Deleted: <#>If *O* is the result of calling the

Deleted: ToString

5. Return the result of calling the [[HasOwnProperty]] internal method of *O* passing *P* as the argument.

NOTE The ordering of steps 1 and 3 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the **this** value is **undefined** or **null**.

Deleted: Let *desc* be

Deleted: GetOwnProperty

Deleted: <#>If *desc* is **undefined**, return **false**.¶
<#>Return **true**.¶
NOTE 1 Unlike [[HasProperty]] (8.12.6), this method does not consider objects in the prototype chain.¶

Deleted: 2

Deleted: 2

15.2.4.6 Object.prototype.isPrototypeOf (V)

When the `isPrototypeOf` method is called with argument *V*, the following steps are taken:

1. If *V* is not an object, return **false**.
2. Let *O* be the result of calling ToObject passing the **this** value as the argument.
3. ReturnIfAbrupt(*O*).
4. Repeat
   a. Let *V* be the result of calling the [[GetInheritance]] internal method of *V* with no arguments.
   b. if *V* is **null**, return **false**
   c. If *O* and *V* refer to the same object, return **true**.

Deleted: value

Deleted: of

Deleted: Prototype

Deleted: e

Deleted: property

NOTE The ordering of steps 1 and 2 is chosen to preserve 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.

15.2.4.7 Object.prototype.propertyIsEnumerable (V)

When the `propertyIsEnumerable` method is called with argument *V*, the following steps are taken:

1. Let *P* be ToString(*V*).
2. ReturnIfAbrupt(*P*).
3. Let *O* be the result of calling ToObject passing the **this** value as the argument.
4. ReturnIfAbrupt(*O*).
5. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* passing *P* as the argument.
6. If *desc* is **undefined**, return **false**.
7. Return the value of *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**.

15.2.5 Properties of Object Instances

Object instances have no special properties beyond those inherited from the Object prototype object.

15.3 Function Objects

15.3.1 The Function Constructor

Deleted: Called as a Function

The Function constructor is the %Function% intrinsic object and the initial value of the **Function** property of the global **object**. When `Function` is called as a function rather than as a constructor, it creates and initialises a new Function object. Thus the function call `Function(…)` is equivalent to the object creation expression `new Function(…)` with the same arguments. However, if the **this** value value passed in the call is an Object with an [[Code]] internal data property whose value is **undefined**, it initialises the **this** value using the argument values. This permits `Function` to be used both as factory method and to perform constructor instance initialization.

Deleted: ize

`Function` may be subclassed and subclass constructors may perform a `super` invocation of the `Function` constructor to initialise subclass instances. However, all syntactic forms for defining function objects create instances of `Function`. There is no syntactic means to create instances of `Function` subclasses except for the built-in Generator Function subclass.

Deleted: ize

15.3.1.1 Function (p1, p2, … , pn, body)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

When the **`Function`** function is called with some arguments *p1*, *p2*, … , *pn*, *body* (where *n* might be 0, that is, there are no "*p*" arguments, and where *body* might also not be provided), the following steps are taken:

1. Let *argCount* be the total number of arguments passed to this function invocation.
2. Let *P* be the empty String.
3. If *argCount* = 0, let *bodyText* be the empty String.
4. Else if *argCount* = 1, let *bodyText* be that argument.
5. Else *argCount* > 1,
    a. Let *firstArg* be the first argument.
    b. Let *P* be ToString(*firstArg*).
    c. ReturnIfAbrupt(*P*).
    d. Let *k* be 2.
    e. Repeat, while *k* < *argCount*
        i. Let *nextArg* be the *k*'th argument.
        ii. Let *nextArgString* be ToString(*nextArg*).
        iii. ReturnIfAbrupt(*nextArgString*).
        iv. Let *P* be the result of concatenating the previous value of *P*, the String **`","`** (a comma), and *nextArgString*.
        v. Increase *k* by 1.
    f. Let *bodyText* be the *k*'th argument.
6. Let *bodyText* be ToString(*bodyText*).
7. ReturnIfAbrupt(*bodyText*).
8. Let *parameters* be the result of parsing *P*, interpreted as UTF-16 encoded Unicode text as described in 8.4, using *FormalParameters* as the goal symbol. Throw a **SyntaxError** exception if the parse fails.
9. Let *body* be the result of parsing *bodyText*, interpreted as UTF-16 encoded Unicode text as described in 8.4, using *FunctionBody* as the goal symbol. Throw a **SyntaxError** exception if the parse fails or if any static semantics errors are detected.
10. If *funcBody* Contains *YieldExpression* is **true**, then throw a **SyntaxError** exception.
11. If IsSimpleParameterList of *parameters* is **false** and any element of the BoundNames of *parameters* also occurs in the VarDeclaredNames of *body*, then throw a **SyntaxError** exception.
12. If any element of the BoundNames of *parameters* also occurs in the LexicallyDeclaredNames of *body*, then throw a **SyntaxError** exception.
13. If *bodyText* is strict mode code (see 10.1.1) then let *strict* be **true**, else let *strict* be **false**.
14. Let *scope* be the Global Environment.
15. Let *F* be the **this** value.
16. If Type(*F*) is not Object or if *F* does not have a [[Code]] internal data property or if the value of [[Code]] is not **undefined**, then
    a. Let *F* be the result of calling FunctionAllocate with agument **`%FunctionPrototype%`**.
17. If the value of *F's* [[FunctionKind]] internal data property is not **`"normal"`**, then throw a **TypeError** exception.
18. Perform the FunctionInitialise abstract operation with arguments *F*, Normal, *parameters*, *body*, *scope*, and *strict*.
19. Perform the abstract operation MakeConstructor with argument *F*.
20. Return *F*.

A **`prototype`** property is automatically created for every function created using the **`Function`** constructor, to provide for the possibility that the function will be used as a constructor.

NOTE It is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
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")
```

Deleted: When the **`Function`** function is called with some arguments *p1*, *p2*, … , *pn*, *b*… (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:¶
<#>Create and return a new Function object as … the standard built-in constructor Function was u… in a **new** expression with the same arguments (15.3.2.1).¶
<#>15.3.2 The Function Constructor¶
When **`Function`** is called as part of a **`new`** expression, it is a constructor: it initialises the newly created object.¶
<#>15.3.2.1 new Function (p1, p2, … , pn, body)¶

Deleted: constructor

Deleted: ,

Deleted: ToString(

Deleted:)

Deleted: If

Deleted: is not parsable as a

Deleted: *List*

Deleted: opt

Deleted: then t

Deleted: <#>If *body* is not parsable as *FunctionBody* then throw a **SyntaxError** exception.¶

Deleted: ize

Deleted: <#>If *strict* is **true**, throw any exceptions specified in 13.1 that apply. ¶

Deleted: <#>Return a new Function object created as specified in 13.2 6 passing *P* *paramet*… as the *FormalParameterList*$_{opt}$ and *body* as the *FunctionBody*. Pass in the Global Environment … the *Scope* parameter and *strict* as the *Strict* flag.…

15.3.1.2 new Function (... argumentsList)

When **`Function`** is called as part of a **`new`** expression, it is a constructor: it initialises the newly created object.

1. Let *F* be the **`Function`** function object on which the **`new`** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **`new`** operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If **`Functiuon`** is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.3.2 Properties of the Function Constructor

The Function constructor is itself a built-in Function object. The value of the [[Prototype]] internal data property of the Function constructor is %FunctionPrototype%, the intrinsic Function prototype object (15.3.3).

The value of the [[Extensible]] internal data property of the Function constructor is **true**.

The Function constructor has the following properties:

15.3.2.1 Function.prototype

The value of **`Function.prototype`** is the %FunctionPrototype%, the intrinsic Function prototype object (15.3.3).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.3.2.2 Function.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.3.2.3 Function[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *proto* be the result of GetPrototypeFromConstructor(*F*, **`"%FunctionPrototype%"`**).
3. ReturnIfAbrupt(*proto*).
4. Let *obj* be the result of calling FunctionAllocate with agument *proto*.
5. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE The Function **`@@create`** function is intentionally generic; it does not require that its **this** value be the Function constructor object. It can be transferred to other constructor functions for use as a **`@@create`** method. When used with other constructors, this function will create a exotic function object whose [[Prototype]] value is obtained from the associated constructor.

15.3.3 Properties of the Function Prototype Object

The Function prototype object is itself a Built-in Function object. When invoked, it accepts any arguments and returns **undefined**.

NOTE The Function prototype object is specified to be a function object to ensure compatability with ECMAScript code that was created prior to the 6th Edition of this specification.

Deleted: The
Deleted: Constructor
Deleted: and its has a [[ClassIsFunctionNativeBrandBuiltinBrand]] is **`"Function"`** internal data property whose value is NativeFunctionBuiltinFunction
Deleted: standard built-in
Deleted: 15.3.4
Deleted: 15.3.3
Deleted: initial
Deleted: standard
Deleted: built-in
Deleted: 15.3.4
Deleted: 15.3.3
Deleted: and has a [[IsFunctionNativeBrandBuiltinBrand]] internal data property whose value is NativeFunctionBuiltinFunction
Deleted: (its [[Class]] is **`"Function"`**) that,
Deleted: when
Formatted: Note

The value of the [[Prototype]] internal data property of the Function prototype object is the intrinsic object %ObjectPrototype% (15.2.4). The initial value of the [[Extensible]] internal data property of the Function prototype object is **true**.

The function prototype object does not have a **`prototype`** property.

The **`length`** property of the Function prototype object is **0**.

15.3.3.1 Function.prototype.constructor

The initial value of **`Function.prototype.constructor`** is the intrinsic object %Function%.

15.3.3.2 Function.prototype.toString ( )

An implementation-dependent representation of the function is returned. This representation has the syntax of a *FunctionDeclaration*. Note in particular that the use and placement of white space, line terminators, and semicolons within the representation String is implementation-dependent.

The **`toString`** function is not generic; it throws a **TypeError** exception if its **this** value is not a Function object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.3.3.3 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:

9. If IsCallable(*func*) is **false**, then throw a **TypeError** exception.
10. If *argArray* is **null** or **undefined**, then
    a. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as *thisArgument* and an empty List of arguments as *argumentsList*.
11. Let *argList* be be the result of CreateListFromArrayLike(*argArray*).
12. ReturnIfAbrupt(*argList*).
13. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as *thisArgument* and *argList* as *argumentsList*.

The **`length`** property of the **`apply`** method is **2**.

NOTE The thisArg value is passed without modification as the **this** value. This is a change from Edition 3, where a **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.

15.3.3.4 15.3.3.4 Function.prototype.call (thisArg [, arg1 [, arg2, …]])

When the **`call`** method is called on an object *func* with argument *thisArg* and optional arguments *arg1*, *arg2* etc, the following steps are taken:

1. If IsCallable(*func*) is **false**, then throw a **TypeError** exception.
2. Let *argList* be an empty List.
3. If this method was called with more than one argument then in left to right order starting with *arg1* append each argument as the last element of *argList*
4. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as *thisArgument* and *argList* as *argumentsList*.

The **`length`** property of the **`call`** method is **1**.

NOTE The thisArg value is passed without modification as the **this** value. This is a change from Edition 3, where a **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.

Deleted: standard built-in

Deleted: p

Deleted: object

Deleted: The Function prototype object does not have a **`valueOf`** property of its own; however, it inherits the **`valueOf`** property fro the Object prototype Object.¶

Deleted: 15.3.4

Deleted: built-in

Deleted: **`Function`** constructor

Deleted: 15.3.4

Deleted: 15.3.4

Deleted: the **this** value

Deleted: list

Deleted: <#>If Type(*argArray*) is not Object then throw a **TypeError** exception.¶
<#>Let *len* be the result of calling the [[Get]] internal method of(*argArray* with argument, **`"length"`**).¶
<#>Let *n* be ToUint32(*len*).¶
<#>ReturnIfAbrupt(*n*).¶

Deleted: an empty List

Deleted: <#>Let *index* be 0.¶
<#>Repeat while *index* < *n*¶
<#>Let *indexName* be ToString(*index*).¶
<#>Let *nextArg* be the result of calling the [[Get]] internal method of(*argArray* with , *indexName* as the argument).¶
<#>ReturnIfAbrupt(*nextArg*).¶
<#>Append *nextArg* as the last element of *argList*.¶
<#>Set *index* to *index* + 1.¶

Deleted: the

Deleted: **this** value

Deleted: *the list of arguments*

Deleted: 15.3.4

Deleted: the **this** value

Deleted: *the list of arguments*

15.3.3.5 Function.prototype.bind (thisArg [, arg1 [, arg2, …]])

The bind method takes one or more arguments, *thisArg* and (optionally) *arg1*, *arg2*, etc, and returns a new function object by performing the following steps:

1. Let *Target* be the **this** value.
2. If IsCallable(*Target*) is **false**, throw a **TypeError** exception.
3. Let *A* be a new (possibly empty) internal list consisting of all of the argument values provided after *thisArg* (*arg1*, *arg2* etc), in order.
4. Let *F* be the result of the abstract operation BoundFunctionCreate with arguments *Target*, *thisArg*, and *A*.
5. If *Target* has a [[BoundTargetFunction]] internal data property, then
   a. Let *targetLen* be the result of Get(*Target*, `"length"`).
   b. ReturnIfAbrupt(*targetLen*).
   c. Let *L* be the larger of 0 and the result of *targetLen* minus the number of elements of *A*.
6. Else let *L* be 0.
7. Call the [[DefineOwnProperty]] internal method of *F* with arguments `"length"` and PropertyDescriptor {[[value]]: *L*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
8. Perform the AddRestrictedFunctionProperties abstract operation with argument *F*.
9. Return *F*.

The `length` property of the `bind` method is **1**.

NOTE Function objects created using `Function.prototype.bind` are exotic objects. They also do not have a `prototype` property.

15.3.3.6 Function.prototype[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Return the result of calling OrdinaryCreateFromConstructor(*F*, `"%ObjectPrototype%"`).

NOTE This is default @@create method that is inherited by all ordinary constructor functions that do not explicitly over-ride it.

15.3.3.7 Function.prototype[@@hasInstance] (V)

When the @@hasInstance method of an object *F* is called with value *V*, the following steps are taken:

1. Let *F* be the **this** value.
2. Return the result of OrdinaryHasInstance(*F*, *V*).

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 deterimine whether a value is an instance of a specific constructor. An expression such as

```
v instanceof F
```

evaluates as

```
F[@@hasInstance](v)
```

A constructor function can control which objects are recognize 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.

15.3.4 Function Instances

Every function instance is an ordinary function object and has the internal data properties listed in Table 13.

Deleted: 15.3.4
Deleted: a new native ECMAScript object
Deleted: <#>Set all the internal methods, except for [[Get]], of *F* as specified in 8.12.¶ <#>Set the [[Get]] internal property of *F* as specified in 15.3.5.4.¶ <#> of *F* to **true**.¶ <#>Set the [[TargetFunction]] internal property of *F* to *Target*.¶ <#>Set the [[BoundThis]] internal property of *F* to the val of *thisArg*.¶ <#>Set the [[BoundArgs]] internal property of *F* to *A*.¶ <#>Set Add the [[ClassIsFunctionNativeBrand]] internal property with value NativeFunction of to *F* to **"Function"**
Deleted: the [[Class]] internal property of
Deleted: is **"Function"**
Deleted: the
Deleted: [[IsFunctionNativeBrandBuiltinBrand]]
Deleted: with value NativeFunctionBuiltinFunction
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: the **length** property of *Target*
Deleted: length
Comment [AWB15145]: TODO This need to be
Deleted: <#>Set the **length** own property of *F* to eith
Deleted: set
Deleted: the **length** own property of *F* to
Deleted: ,
Deleted: , and **false**
Deleted: or
Deleted: <#>Set the attributes of the **length** own prope
Deleted: <#>Set the [[Extensible]] internal property of
Deleted: <#>Let *thrower* be the [[ThrowTypeError]]
Deleted: or the [[Code]], [[FormalParameters]], and
Deleted: 15.3.4
Deleted: When the @@create method of an object *F*,
Formatted: Note
Deleted: <#>15.3.4.5.1 [[Call]]¶
Deleted: 15.3.4
Deleted: ;
Deleted: <#>15.3.4.5.3 [[HasInstance]] (V)¶
Deleted: 15.3.5Properties of
Deleted: In addition to the required internal properties
Deleted: a [[Call]] internal property method and in mos
Deleted: property

Function instances that correspond to strict mode functions and function instances created using the **Function.prototype.bind method** (15.3.3.5) have properties named `caller` and `arguments` that throw a **TypeError** exception. An ECMAScript implementation must not associate any implementation specific behaviour with accesses of these properties from strict mode function code.

The Function instances have the following properties:

15.3.4.1 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]]: **false** }.

15.3.4.2 prototype

Function instances that can be used used as a constructor have a `prototype` property. Whenever such a function instance is created another ordinary object is also created and is the initial value of the function's `prototype` property. Unless otherwise specified, the value of the prototype property is used to initialise the [[Prototype]] internal data property of a newly created ordinay object before the Function object is invoked as a constructor for that newly created object.

This property has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE Function objects created using `Function.prototype.bind` or by evaluating a *MethodDefinition* (that is not a *GeneratorMethod*) or an *ArrowFunction* grammar production do not have a `prototype` property.

15.4 Array Objects

Array objects are exotic objects that give special treatment to a certain class of property names. See 8.4.3 for a definition of this special treatment.

An Array object, *O*, is said to be *sparse* if the following algorithm returns **true**:

1. Let *len* be the result of Get(*O*, **"length"**).
2. For each integer *i* in the range 0≤*i*<ToUint32(*len*)
   a. Let *elem* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument ToString(*i*).
   b. If *elem* is **undefined**, return **true**.
3. Return **false**.

15.4.1 The Array Constructor

The Array constructor is the %Array% instrinsic object and the initial value of the `Array` property of the global object. When `Array` is called as a function rather than as a constructor, it creates and initialises a new Array object. Thus the function call `Array(…)` is equivalent to the object creation expression `new Array(…)` with the same arguments. However, if the **this** value value passed in the call is an Object with an [[ArrayInitialisationState]] internal data property whose value is **undefined**, it initialises the **this** value using the argument values. This permits `Array` to be used both as factory method and to perform constructor instance initialization.

The `Array` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified `Array` behaviour must include a `super` call to the `Array` constructor to initialise subclass instances.

Deleted: The value of the [[Class]] internal property is **"Function"**.Every function instanc has a [[IsFunctionNativeBrandBuiltinBrand]] internal data property whose value is NativeFunctionBuiltinFunction.¶
Deleted: (13.26)
Deleted: 15.3.4
Deleted: "
Deleted: "
Deleted: "
Deleted: "
Deleted: 15.3.5
Deleted: "
Deleted: "
Deleted: 15.3.5
Deleted: The
Deleted: .
Deleted:
Deleted: <#>15.3.5.3 [[HasInstance]] Assume *F* is a Function object.¶
When the [[HasInstance]] internal method of is called with value *V*, the following steps are taken:¶
<#>If *V* is not an object, return **false**.¶
<#>Let *O* be the result of calling the [[Get]] internal method of *F* with property name **"prototype"**.¶
<#>ReturnIfAbrupt(*O*).¶
<#>If Type(*O*) is not Object, throw a **TypeErr** exception.¶
<#>Repeat¶
<#>Let *V* be the value of the [[Prototype]] internal property of *V*.¶
<#>If *V* is **null**, return **false**.¶
<#>If *O* and *V* refer to the same object, return **true**.¶
NOTE Function objects created using `Function.prototype.bind` have a differe implementation of [[HasInstance]] defined in 15.3.4.5.3.¶
Deleted: 2
Deleted: A property name *P* (in the form of a String value) is an *array index* if and only if
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Comment [AWB11146]: TODO: see if this algorithm is really needed
Deleted: <#>The Array Constructor
Deleted: ¶
Deleted: z
Deleted: ize
Deleted: `Array` may be subclassed and
Deleted: ize

15.4.1.1 Array ([item1 [, item2 [, …]]])

This description applies if and only if the Array constructor is called with no arguments or at least two arguments.

When the **Array** function is called the following steps are taken:

1. Let *numberOfArgs* be the number of arguments passed to this constructor call.
2. Assert: *numberOfArgs* ≠ 1.
3. Let *O* be the **this** value.
4. If Type(*O*) is Object and *O* has a [[ArrayInitialisationState]] internal data property and the value of [[ArrayInitialisationState]] is **false**, then
   a. Set the value of *O's* [[ArrayInitialisationState]] internal data property to **true**.
   b. Let *array* be *O*.
5. Else,
   a. Let *F* be this function.
   b. Let *proto* be the result of GetPrototypeFromConstructor(*F*, `"%ArrayPrototype%"`).
   c. ReturnIfAbrupt(*proto*).
   d. Let *array* be the result of the abstract operation ArrayCreate with arguments *numberOfArgs* and *proto*.
6. ReturnIfAbrupt(*array*).
7. Let *k* be 0.
8. Let *items* be a zero-origined List contain the argument items in order.
9. Repeat, while *k* < *numberOfArgs*
   a. Let *Pk* be ToString(*k*).
   b. Let *itemK* be *k*th element of *items*.
   c. Let *defineStatus* be the result of DefinePropertyOrThrow(*array*, *Pk*, Property Descriptor {[[Value]]: *itemK*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
   d. ReturnIfAbrupt(*defineStatus*).
   e. Increase *k* by 1.
10. Let *putStatus* be the result of Put(*array*, `"length"`, *numberOfArgs*, **true**).
11. ReturnIfAbrupt(*putStatus*).
12. Return *array*.

15.4.1.2 Array (len)

This description applies if and only if the Array constructor is called with exactly one argument.

1. Let *numberOfArgs* be the number of arguments passed to this constructor call.
2. Assert: *numberOfArgs* = 1.
3. Let *O* be the **this** value.
4. If Type(*O*) is Object and *O* has an [[ArrayInitialisationState]] internal data property and the value of [[ArrayInitialisationState]] is **false**, then
   a. Set the value of *O's* [[ArrayInitialisationState]] internal data property to **true**.
   b. Let *array* be *O*.
5. Else,
   a. Let *F* be this function.
   b. Let *proto* be the result of GetPrototypeFromConstructor(*F*, `"%ArrayPrototype%"`).
   c. ReturnIfAbrupt(*proto*).
   d. Let *array* be the result of the abstract operation ArrayCreate with argument 0 and *proto*.
6. ReturnIfAbrupt(*array*).
7. If Type(*len*) is not Number, then
   a. Let *defineStatus* be the result of DefinePropertyOrThrow(*array*, `"0"`, Property Descriptor {[[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
   b. ReturnIfAbrupt(*defineStatus*).
   c. Let *intLen* be 1.
8. Else,
   a. Let *intLen* be ToUint32(*len*).
   b. If *intLen* ≠ *len*, then throw a **RangeError** exception.

Deleted: given
Deleted: <#>Create and return a new Array object exac as if the standard built-in constructor **Array** was used in a **new** expression with the same arguments (15.4.2).Return t
Deleted: *len*
Deleted: z
Deleted: z
Deleted: z
Deleted: of
Deleted: abstraction operation
Deleted: (15.4)
Deleted: *len*
Deleted: *len*
Deleted: Let *defineStatus* be the result of calling
Deleted: Call the [[
Deleted: Own
Deleted:]]
Deleted: internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: , and **true**
Deleted: <#>ReturnIfAbrupt(*defineStatus*).¶
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: *len*
Deleted: and
Deleted: <#>The [[Prototype]] internal property
Deleted: new
Deleted: given
Deleted: z
Deleted: z
Deleted: z
Deleted: *s*
Deleted: The [[Prototype]] internal property of the new
Deleted: <#>Let *array* be the result of of the abstract
Deleted: calling the [[
Deleted: Own
Deleted:]]
Deleted: internal method of
Deleted: with
Deleted: arguments
Deleted: , and **true**
Deleted: Return *array*.

9. Let *putStatus* be the result of Put(*array*, `"length"`, *intLen*, **true**).
10. ReturnIfAbrupt(*putStatus*).
11. Return *array*.

15.4.1.3 new Array (... argumentsList)

When `Array` is called as part of a `new` expression, it initialises a newly created object.

1. Let *F* be the `Array` function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If `Array` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.4.2 Properties of the Array Constructor

The value of the [[Prototype]] internal data property of the Array constructor is the Function prototype object (15.3.3).

Besides and the `length` property (whose value is **1**), the Array constructor has the following properties:

15.4.2.1 Array.prototype

The value of `Array.prototype` is %ArrayPrototype% the intrinsic Array prototype object (15.4.3).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.4.2.2 Array.isArray (arg)

The isArray function takes one argument *arg*, and performs the following steps are taken:

1. If Type(*arg*) is not Object, return **false**.
2. If *arg* is an exotic Array object, then return **true**.
3. Return **false**.

15.4.2.3 Array.of (...items)

When the `of` method is called with any number of arguments, the following steps are taken:

1. Let *lenValue* be the result of Get(*items*, `"length"`).
2. Let *len* be ToInteger(*lenValue*).
3. Let *C* be the **this** value.
4. If IsConstructor(*C*) is **true**, then
   a. Let *newObj* be the result of calling the [[Construct]] internal method of *C* with an argument list containing the single item *len*.
   b. Let *A* be ToObject(*newObj*).
5. Else,
   a. Let *A* be the result of the abstract operation ArrayCreate with argument *len*.
6. ReturnIfAbrupt(*A*).
7. Let *k* be 0.
8. Repeat, while *k* < *len*
   a. Let *Pk* be ToString(*k*).
   b. Let *kValue* be the result of Get(*items*, *Pk*).
   c. Let *defineStatus* be the result of DefinePropertyOrThrow(*A*,*Pk*, Property Descriptor {[[Value]]: *kValue*.[[value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
   d. ReturnIfAbrupt(*defineStatus*).
   e. Increase *k* by 1.

Deleted: <#>Let *array* be the result of of the abstraction operationabstract operation ArrayCr (15.4) with argument *intLen*.¶
Deleted: The
Deleted: Constructor
Deleted: it is a constructor:
Deleted: the
Deleted: be the be the
Deleted: a
Deleted: 15.3.4
Deleted: 15.3.3
Deleted: the internal properties
Deleted: initial
Deleted: 15.4.4
Deleted: returns the Boolean value **true** if th argument is an object whose class [[BuiltinBrand]] internal property is "Array"BuildinArray; otherwise it returns **false** The
Deleted: the value of the [[Class]] internal property of
Deleted: is `"Array"`
Deleted: has the [[IsArrayNativeBrandBuiltinBrand]] internal property with value NativeArrayBuiltinArray
Deleted: ...
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with the argument
Deleted: i
Comment [AWB7147]: It would be nice to have a more explicit way to create a collectio with a pre-specified number of elements.
Deleted: abstraction operation
Deleted: (15.4)
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: of calling the [[
Deleted: Own
Deleted:]] internal method of
Deleted: with arguments
Deleted: , and **true**

9. Let *putStatus* be the result of Put(*A*, `"length"`, *len*, **true**).
10. ReturnIfAbrupt(*putStatus*).
11. Return *A*.

The `length` property of the `of` method is **0**.

NOTE 1 The *items* argument is assume to be a well-formed rest argument value.

NOTE 2 The `of` function is an intentionally generic factory method; it does not require that its **this** value be the Array constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument.

15.4.2.4 Array.from (arrayLike , mapfn=undefined, thisArg=undefined)

When the `from` method is called with argument *arrayLike* and optional arguments *mapfn* and *thisArg* the following steps are taken:

1. Let *C* be the **this** value.
2. Let *items* be ToObject(*arrayLike*).
3. ReturnIfAbrupt(*items*).
4. If *mapfn* is **undefined**, then let *mapping* be **true.**
5. else
   a. If IsCallable(*mapfn*) is **false**, throw a **TypeError** exception.
   b. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
   c. Let *mapping* be **true**
6. Let *usingIterator* be the result of HasProperty(*items*, @@Iterator).
7. ReturnIfAbrupt(*usingIterator*).
8. If *usingIterator* is **true**, then
   a. Let *iterator* be the result of performing GetIterator(*obj*).
   b. ReturnIfAbrupt(*iterator*).
   c. If IsConstructor(*C*) is **true**, then
      i. Let *newObj* be the result of calling the [[Construct]] internal method of *C* with an empty argument list.
      ii. Let *A* be ToObject(*newObj*).
   d. Else,
      i. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
   e. ReturnIfAbrupt(*A*).
   f. Let *done* be **false**.
   g. Let *k* be 0.
   h. Repeat, while *done* is **false**
      i. Let *Pk* be ToString(*k*).
      ii. Let *next* be the result of IteratorNext(*iterator*).
      iii. ReturnIfAbrupt(*next*).
      iv. Let *done* be IteratorComplete(*next*).
      v. ReturnIfAbrupt(*done*).
      vi. If *done* is **true**, then
         1. Let *putStatus* be the result of Put(*A*, `"length"`, *k*, **true**).
         2. ReturnIfAbrupt(*putStatus*).
         3. Return *A*.
      vii. Let *nextValue* be IteratorValue(*next*).
      viii. ReturnIfAbrupt(*nextValue*).
      ix. If *mapping* is **true**, then
         1. Let *mappedValue* be the result of calling the [[Call]] internal method of *mapfn* with *T* as *thisArgument* and a List containing *kValue* as *argumentsList*.
         2. ReturnIfAbrupt(*mappedValue*).
      x. Else, let *mappedValue* be *kValue*.
      xi. Let *defineStatus* be the result of DefinePropertyOrThrow(*A*, *Pk* Property Descriptor {[[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
      xii. ReturnIfAbrupt(*defineStatus*).
      xiii. Increase *k* by 1.

Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: `with arguments`
Deleted: and
Deleted: any number of
Deleted: s
Deleted: ,
Moved (insertion) [7]
Comment [AWB7148]: It would be nice to have a m explicit way to create a collection with a pre-specified number of elements.

9. Assert: *items* is not an Iterator so assume it is Array-like.
10. Let *lenValue* be the result of Get(*items*, `"length"`).
11. Let *len* be ToInteger(*lenValue*).
12. ReturnIfAbrupt(*len*).
13. If IsConstructor(*C*) is **true**, then
    a. Let *newObj* be the result of calling the [[Construct]] internal method of *C* with an argument list containing the single item *len*.
    b. Let *A* be ToObject(*newObj*).
14. Else,
    a. Let *A* be the result of the abstract operation ArrayCreate with argument *len*.
15. ReturnIfAbrupt(*A*).
16. Let *k* be 0.
17. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*items*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*items*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. If *mapping* is **true**, then
            1. Let *mappedValue* be the result of calling the [[Call]] internal method of *mapfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *items* as *argumentsList*.
            2. ReturnIfAbrupt(*mappedValue*).
        iv. Else, let *mappedValue* be *kValue*.
        v. Let *defineStatus* be the result of DefinePropertyOrThrow(*A*, *Pk*, Property Descriptor {[[Value]]: *mappedValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
        vi. ReturnIfAbrupt(*defineStatus*).
    e. Increase *k* by 1.
18. Let *putStatus* be the result of Put(*A*, `"length"`, *len*, **true**).
19. ReturnIfAbrupt(*putStatus*).
20. Return *A*.

NOTE The `from` function is an intentionally generic factory method; it does not require that its **this** value be the Array constructor. Therefore it can be transferred to or inherited by any other constructors that may be called with a single numeric argument.

15.4.2.5 Array[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *proto* be the result of GetPrototypeFromConstructor(*F*, `"%ArrayPrototype%"`).
3. ReturnIfAbrupt(*proto*).
4. Let *obj* be the result of calling ArrayCreate with arguments **undefined** and *proto*.
5. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE 1 Passing **undefined** as the first argument to ArrayCreate causes the [[ArrayInitialisationState]] internal data property of the array to be initially assigned the value **false**. This is a flag used to indicate that the instance has not yet been initialised by the `Array` constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in any unobservable manner.

NOTE 2 The Array `@@create` function is intentionally generic; it does not require that its **this** value be the Array constructor object. It can be transferred to other constructor functions for use as a `@@create` method. When used with other constructors, this function will create an exotic Array object whose [[Prototype]] value is obtained from the associated constructor.

Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with the argument
Deleted: ¶
Moved up [7]: Let *C* be the **this** value.¶
Deleted: i
Comment [AWB7149]: It would be nice to have a more explicit way to create a collectio with a pre-specified number of elements.
Deleted: abstraction operation
Deleted: (15.4)
Deleted: calling the [[HasProperty]] interna method of
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: calling the [[
Deleted: Own
Deleted:]] internal method of
Deleted: with arguments
Deleted: ,
Deleted: and
Deleted: *kValue*.[[value]]
Deleted: , and **true**
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: false
Deleted: This method may be assigned to ¶
Deleted: z
Deleted: z

15.4.3 Properties of the Array Prototype Object

The value of the [[Prototype]] internal data property of the Array prototype object is the intrinsic object %ObjectPrototype%.

The Array prototype object is itself an ordinary object. It is not an Array instance and does not have a **`length`** property.

In following descriptions of functions that are properties of the Array prototype object, the phrase "this object" refers to the object that is the **this** value for the invocation of the function. It is permitted for the **this** object to be an object which is not an exotic Array object.

NOTE The Array prototype object does not have a **`valueOf`** property of its own; however, it inherits the **`valueOf`** property from the standard built-in Object prototype Object.

15.4.3.1 Array.prototype.constructor

The initial value of **`Array.prototype.constructor`** is the standard built-in **`Array`** constructor.

15.4.3.2 Array.prototype.toString ( )

When the **`toString`** method is called, the following steps are taken:

1. Let *array* be the result of calling ToObject on the **this** value.
2. ReturnIfAbrupt(*array*).
3. Let *func* be the result of Get(*array*, **`"join"`**).
4. ReturnIfAbrupt(*func*).
5. If IsCallable(*func*) is **false**, then let *func* be the standard built-in method Object.prototype.toString (15.2.4.2).
6. Return the result of calling the [[Call]] internal method of *func* providing *array* as *thisArgument* and an empty List as *argumentsList*.

NOTE The **`toString`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`toString`** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.3 Array.prototype.toLocaleString ( )

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.

The result is calculated as follows:

1. Let *array* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*array*).
3. Let *arrayLen* be the result of Get(*array*, **`"length"`**).
4. Let *len* be ToUint32(*arrayLen*).
5. ReturnIfAbrupt(*len*).
6. Let *separator* be the String value for the list-separator String appropriate for the host environment's current locale (this is derived in an implementation-defined way).
7. If *len* is zero, return the empty String.
8. Let *firstElement* be the result of Get(*array*, **`"0"`**).
9. Let *noArgs* be an empty List.
10. ReturnIfAbrupt(*firstElement*).
11. If *firstElement* is **undefined** or **null**, then
    a. Let *R* be the empty String.
12. Else
    a. Let *R* be the result of Invoke(*firstElement*, **`"toLocaleString"`**).

Deleted: standard built-in Object prototype object (15.2.4)

Deleted: array

Deleted: ; its [[Class]] is **`"Array"`** it has an [[NativeBrandBuiltinBrandIsArray]] internal data proper with value NativeArrayBuiltinArray

Deleted: and it has

Deleted: (whose initial value is **+0**) and the special [[DefineOwnProperty]] internal method described in 15.4.5.1

Deleted: for

Deleted: the value of the [[Class]] internal property is **`"Array"`**

Deleted: does not have an [[IsArrayNativeBrandBuiltinBrand]] internal data proper with value NativeArrayBuiltinArray

Deleted: If *array* is an abrupt completion, return *array*.

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: If *func* is an abrupt completion, return *func*.

Deleted: the **this** value

Deleted: *l*

Deleted: host

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: calling the [[

Deleted:]]

Deleted: internal method of

Deleted: with argument

Deleted: <#>Let *elementObj* be ToObject(*firstEleme*
<#>Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument **`"toLocaleString"`**.¶
<#>If IsCallable(*func*) is **false**, throw a **TypeError** exception.¶

Deleted: calling

Deleted: with arguments

Deleted: , *firstElement*,

Deleted: and an empty arguments List

b. Let *R* be ToString(*R*).
c. ReturnIfAbrupt(*R*).

13. Let *k* be **1**.
14. Repeat, while *k* < *len*
    a. Let *S* be a String value produced by concatenating *R* and *separator*.
    b. Let *nextElement* be the result of Get(*array*, ToString(*k*)).
    c. ReturnIfAbrupt(*nextElement*).
    d. If *nextElement* is **undefined** or **null**, then
        i. Let *R* be the empty String.
    e. Else
        i. Let *R* be the result of Invoke(*nextElement*, `"toLocaleString"`).
        ii. Let *R* be ToString(*R*).
        iii. ReturnIfAbrupt(*R*).
    f. Let *R* be a String value produced by concatenating *S* and *R*.
    g. Increase *k* by 1.
15. Return *R*.

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleString` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `toLocaleString` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.4 Array.prototype.concat ([item1 [, item2 [, …]]])

When the `concat` method is called with zero or more arguments *item1*, *item2*, etc., it returns an array containing the array elements of the object followed by the array elements of each argument in order.

The following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. Let *A* be **undefined**.
3. If *O* is an exotic Array object, then
    a. Let *C* be the Get(*O*, `"constructor"`).
    b. ReturnIfAbrupt(*C*).
    c. If IsConstructor(*C*) is **true**, then
        i. Let *newObj* be the result of calling the [[Construct]] internal method of *C* with an argument list containing the single item 0.
        ii. Let *A* be ToObject(*newObj*).
4. If *A* is **undefined**, then
    a. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
5. ReturnIfAbrupt(*A*).
6. Let *n* be 0.
7. Let *items* be an internal 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.
8. Repeat, while *items* is not empty
    a. Remove the first element from *items* and let *E* be the value of the element.
    b. Let *spreadable* be the result of IsConcatSpreadable(*E*).
    c. ReturnIfAbrupt(*spreadable*).
    d. If *spreadable* is **true**, then
        i. Let *k* be 0.
        ii. Let *len* be the result of Get(*E*, `"length"`).
        iii. ReturnIfAbrupt(*len*).
        iv. Repeat, while *k* < *len*
            1. Let *P* be ToString(*k*).
            2. Let *exists* be the result of HasProperty(*E*, *P*).
            3. ReturnIfAbrupt(*exists*).
            4. If *exists* is **true**, then

Comment [AWB7150]: This step was missing in ES<=5.1

Deleted: If *R* is an abrupt completion, return *R*.

Deleted: Let *R* be the result of calling the [[Call]] internal method of *func* providing *elementObj* as the **this** value and an empty arguments list.

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: Let *elementObj* be ToObject(*nextElement*).

Deleted: calling

Deleted: with arguments

Deleted: , *nextElement*,

Deleted: and an empty arguments List

Comment [AWB7151]: This step was missing in ES<=5.1

Deleted: If *R* is an abrupt completion, return *R*.

Deleted: Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument `"toLocaleString"`.
If IsCallable(*func*) is **false**, throw a **TypeError** exception.
Let *R* be the result of calling the [[Ca… internal method of *func* providing *element*… as the **this** value and an empty arguments l…

Deleted: host

Deleted: 15.4.4

Deleted: ReturnIfAbrupt(*O*).

Comment [AWB7152]: It would be nice to have a more explicit way to create a collectio… with a pre-specified number of elements.

Deleted: Let *A* be the result of the abstraction operationabstract operation ArrayCr… (15.4) with argument 0a new array created as if the expression `new Array()` where `Array` is the standard built-in constructor with that name.
Let *n* be 0.

Deleted: the value of the [[Class]] internal property of

Deleted: *E*

Deleted: is `"Array"`

Deleted: has

Deleted: an the [[IsArrayNativeBrandBuiltinBrand]] inter… data property with value NativeArrayBuiltinArrayexotic Array obje…

Deleted: calling the [[Get]] internal meth…

Deleted: calling the [[HasProperty]] internal method of

Deleted: with

   a Let *subElement* be the result of Get(*E*, *P*).
   b Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *subElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
  5. Increase *n* by 1.
  6. Increase *k* by 1.

e. Else *E* is not an Array,
  i. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *E*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
  ii. Increase *n* by 1.

9. Let *putStatus* be the result of Put(*A*, `"length"`, *n*, **true**).
10. ReturnIfAbrupt(*putStatus*).
11. Return *A*.

The `length` property of the `concat` method is **1**.

NOTE 1 The explicit setting of the `length` property in step 9 is necessary to ensure that its value is correct in situations where the trailing elements of the result Array are not present.

NOTE 2 The `concat` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `concat` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.4.1 IsConcatSpreadable (O) Abstract Operation

The abstract operation HasArrayConstructor with argument *O* performs the following steps:

1. ReturnIfAbrupt(*O*).
2. Let *spreadable* be the Get(*O*, @@isConcatSpreadable).
3. ReturnIfAbrupt(*spreadable*).
4. If *spreadable* is not **undefined**, then return ToBoolean(*spreadable*).
5. If *O* is an exotic Array object, then return **true**.
6. Return **false**.

15.4.3.5 Array.prototype.join (separator)

The elements of the array are converted to Strings, and these Strings are then concatenated, separated by occurrences of the *separator*. If no separator is provided, a single comma is used as the separator.

The `join` method takes one argument, *separator*, and performs the following steps:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *separator* is **undefined**, let *separator* be the single-character String `","`.
7. Let *sep* be ToString(*separator*).
8. If *len* is zero, return the empty String.
9. Let *element0* be the result of Get(*O*, `"0"`).
10. If *element0* is **undefined** or **null**, let *R* be the empty String; otherwise, let *R* be ToString(*element0*).
11. ReturnIfAbrupt(*R*).
12. Let *k* be `1`.
13. Repeat, while *k* < *len*
    a. Let *S* be the String value produced by concatenating *R* and *sep*.
    b. Let *element* be the result of Get(*O*, ToString(*k*)).
    c. If *element* is **undefined** or **null**, then let *next* be the empty String; otherwise, let *next* be ToString(*element*).

- **Deleted:** calling the [[
- **Deleted:**]] internal method of
- **Deleted:**
- **Deleted:** with argument
- **Deleted:**),
- **Deleted:** , and **false**
- **Deleted:** ,
- **Deleted:**),
- **Deleted:** , and **false**
- **Deleted:** Call
- **Deleted:** calling the [[Put]] internal method of *A* with arguments `"length"`, *n*, and **true**
- **Deleted:** If *putStatus* is an abrupt completion, return *putStatus*.
- **Deleted:** 7
- **Deleted:** host
- **Deleted:** 15.4.4
- **Deleted:** calling the [[Get]] internal method of *O* with argument `"length"`
- **Deleted:** calling the [[
- **Deleted:**]] internal method of
- **Deleted:**
- **Deleted:** with argument
- **Deleted:** Let
- **Deleted:** If *R* is an abrupt completion, return *R*
- **Deleted:** calling the [[
- **Deleted:**]] internal method of
- **Deleted:**
- **Deleted:** with argument
- **Deleted:** L

d. ReturnIfAbrupt(*next*).
e. Let *R* be a String value produced by concatenating *S* and *next*.
f. Increase *k* by 1.
14. Return *R*.

The **length** property of the **join** method is **1**.

NOTE The **join** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **join** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.6 Array.prototype.pop ( )

The last element of the array is removed from the array and returned.

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *len* is zero,
   a. Let *putStatus* be the result of Put(*O*, **"length"**, 0, **true**).
   b. ReturnIfAbrupt(*putStatus*).
   c. Return **undefined**.
7. Else *len* > 0,
   a. Let *newLen* be *len*–1.
   b. Let *indx* be ToString(*newLen*).
   c. Let *element* be the result of Get(*O*, *indx*).
   d. ReturnIfAbrupt(*element*).
   e. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *indx*).
   f. ReturnIfAbrupt(*deleteStatus*).
   g. Let *putStatus* be the result of Put(*O*, **"length"**, *newLen*, **true**).
   h. ReturnIfAbrupt(*putStatus*).
   i. Return *element*.

NOTE The **pop** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **pop** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.7 Array.prototype.push ([item1 [, item2 [, …]]])

The arguments are appended to the end of the array, in the order in which they appear. The new length of the array is returned as the result of the call.

When the **push** method is called with zero or more arguments *item1*, *item2*, etc., the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, **"length"**).
4. Let *n* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*n*).
6. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
7. Repeat, while *items* is not empty
   a. Remove the first element from *items* and let *E* be the value of the element.
   b. Let *putStatus* be the result of Put(*O*, ToString(*n*), *E*, **true**).
   c. ReturnIfAbrupt(*putStatus*).
   d. Increase *n* by 1.
8. Let *putStatus* be the result of Put(*O*, **"length"**, *n*, **true**).

Deleted: If *next* is an abrupt completion, re *next*.
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method of with argument **"length"**
Deleted: Call
Deleted: calling the [[Put]] internal method *O* with arguments **"length"**, 0, and **true**
Deleted: If *putStatus* is an abrupt completic return *putStatus*.
Deleted: ,
Deleted: *indx*
Deleted: ToString(
Deleted:)
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *element* is an abrupt completion return *element*.
Deleted: of calling Call the [[
Deleted:]] internal method of
Deleted: with arguments
Deleted: and **true**
Deleted: If *deleteStatus* is an abrupt completion, return *deleteStatus*.
Deleted: calling Call the [[Put]] internal method of *O* with arguments **"length"**, *indxnewLen*, and **true**
Deleted: If *putStatus* is an abrupt completic return *putStatus*.
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method of with argument **"length"**
Deleted: *len*
Deleted: calling Call the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: If *putStatus* is an abrupt completic return *putStatus*.
Deleted: calling Call the [[Put]] internal metho of *O* with arguments **"length"**, *n*, and **true**

9. ReturnIfAbrupt(*putStatus*).
10. Return *n*.

The **length** property of the **push** method is **1**.

NOTE The **push** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **push** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.8 Array.prototype.reverse ( )

The elements of the array are rearranged so as to reverse their order. The object is returned as the result of the call.

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. Let *middle* be floor(*len*/2).
7. Let *lower* be **0**.
8. Repeat, while *lower* ≠ *middle*
   a. Let *upper* be *len*− *lower* −1.
   b. Let *upperP* be ToString(*upper*).
   c. Let *lowerP* be ToString(*lower*).
   d. Let *lowerValue* be the result of Get(*O*, *lowerP*).
   e. ReturnIfAbrupt(*lowerValue*).
   f. Let *upperValue* be the result of Get(*O*, *upper*).
   g. ReturnIfAbrupt(*upperValue*).
   h. Let *lowerExists* be the result of HasProperty(*O*, *lowerP*).
   i. ReturnIfAbrupt(*lowerExists*).
   j. Let *upperExists* be the result of HasProperty(*O*, *upperP*).
   k. ReturnIfAbrupt(*upperExists*).
   l. If *lowerExists* is **true** and *upperExists* is **true**, then
      i. Let *putStatus* be the result of Put(*O*, *lowerP*, *upperValue*, **true**).
      ii. ReturnIfAbrupt(*putStatus*).
      iii. Let *putStatus* be the result of Put(*O*, *upperP*, *lowerValue*, **true**).
      iv. ReturnIfAbrupt(*putStatus*).
   m. Else if *lowerExists* is **false** and *upperExists* is **true**, then
      i. Let *putStatus* be the result of Put(*O*, *lowerP*, *upperValue*, **true**).
      ii. ReturnIfAbrupt(*putStatus*).
      iii. Let *deleteStatus* be the result of DeletePropertyOrThrow (*O*, *upperP*).
      iv. ReturnIfAbrupt(*deleteStatus*).
   n. Else if *lowerExists* is **true** and *upperExists* is **false**, then
      i. Let *deleteStatus* be the result of DeletePropertyOrThrow (*O*, *lowerP*).
      ii. ReturnIfAbrupt(*deleteStatus*).
      iii. Let *putStatus* be the result of Put(*O*, *upperP*, *lowerValue*, **true**).
      iv. ReturnIfAbrupt(*putStatus*).
   o. Else both *lowerExists* and *upperExists* are **false**,
      i. No action is required.
   p. Increase *lower* by 1.
9. Return *O* .

NOTE The **reverse** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **reverse** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.9 Array.prototype.shift ( )

The first element of the array is removed from the array and returned.

Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method of *O* with argument **"length"**
Deleted: calling the [[…et]] internal method of… ...
Deleted: If *lowerValue* is an abrupt completion, return *lowerValue*
Deleted: calling the [[…et]] internal method of… ...
Deleted: If *upperValue* is an abrupt completion, return *upperValue*
Deleted: calling the [[HasProperty]] internal method
Deleted: calling the [[HasProperty]] internal method
Deleted: calling Call the [[…ut]] internal method
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: calling Call the [[…ut]] internal method
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: calling Call the [[…ut]] internal method
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: calling
Deleted: Call
Deleted: the [[…eletePropertyOrThrow (]] internal
Deleted: If *deleteStatus* is an abrupt completion, return *deleteStatus*
Deleted: calling Call the [[…elete]] internal method
Deleted: If *deleteStatus* is an abrupt completion, return *deleteStatus*
Deleted: calling Call the [[…ut]] internal method
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: ,
Deleted: host
Deleted: 15.4.4

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *len* is zero, then
   a. Let *putStatus* be the result of Put(*O*, `"length"`, 0, **true**).
   b. ReturnIfAbrupt(*putStatus*).
   c. Return **undefined**.
7. Let *first* be the result of Get(*O*, `"0"`).
8. ReturnIfAbrupt(*first*).
9. Let *k* be 1.
10. Repeat, while *k* < *len*
    a. Let *from* be ToString(*k*).
    b. Let *to* be ToString(*k*–1).
    c. Let *fromPresent* be the result of HasProperty(*O*, *from*).
    d. ReturnIfAbrupt(*fromPresent*).
    e. If *fromPresent* is **true**, then
       i. Let *fromVal* be the result of Get(*O*, *from*).
       ii. ReturnIfAbrupt(*fromVal*).
       iii. Let *putStatus* be the result of Put(*O*, *to*, *fromVal*, **true**).
       iv. ReturnIfAbrupt(*putStatus*).
    f. Else *fromPresent* is **false**,
       i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
       ii. ReturnIfAbrupt(*deleteStatus*).
    g. Increase *k* by 1.
11. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, ToString(*len*–1)).
12. ReturnIfAbrupt(*deleteStatus*).
13. Let *putStatus* be the result of Put(*O*, `"length"`, *len*–1, **true**).
14. ReturnIfAbrupt(*putStatus*).
15. Return *first*.

NOTE The `shift` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `shift` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.10 Array.prototype.slice (start, end)

The `slice` method takes two arguments, *start* and *end*, and returns an array containing the elements of the array from element *start* up to, but not including, element *end* (or through the end of the array if *end* is **undefined**). If *start* is negative, it is treated as *length*+*start* where *length* is the length of the array. If *end* is negative, it is treated as *length*+*end* where *length* is the length of the array. The following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
4. Let *lenVal* be the result of Get(*O*, `"length"`).
5. Let *len* be ToUint32(*lenVal*).
6. ReturnIfAbrupt(*len*).
7. Let *relativeStart* be ToInteger(*start*).
8. ReturnIfAbrupt(*relativeStart*).
9. If *relativeStart* is negative, let *k* be max((*len* + *relativeStart*),0); else let *k* be min(*relativeStart*, *len*).
10. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ToInteger(*end*).
11. ReturnIfAbrupt(*relativeEnd*).
12. If *relativeEnd* is negative, let *final* be max((*len* + *relativeEnd*),0); else let *final* be min(*relativeEnd*, *len*).
13. Let *n* be 0.
14. Repeat, while *k* < *final*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).

Deleted: calling the [[Get]] internal method of with argument `"length"`
Deleted: calling Call the [[Put]] internal method of *O* with arguments `"length"`, 0, **true**
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: calling the [[…et]] internal method o
Deleted: If *first* is an abrupt completion, return *first*
Deleted: calling the [[HasProperty]] internal method of ... with argument
Deleted: calling the [[…et]] internal
Deleted: If *fromVal* is an abrupt completion, return *fromVal*
Deleted: calling Call the [[…ut]] interna
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: ,
Deleted: calling Call the [[…elete]] inter
Deleted: If *deleteStatus* is an abrupt completion, return *deleteStatus*
Deleted: calling Call the
Deleted: If *deleteStatus* is an abrupt completion, return *deleteStatus*
Deleted: calling Call the [[…ut]] internal meth
Deleted: If *putStatus* is an abrupt completion, return *putStatus*
Deleted: host
Deleted: 15.4.4
Deleted: abstraction operation
Deleted: (15.4)
Deleted: a new array created as if by the expression `new Array()` where `Array` is th standard built-in constructor with that name
Deleted: calling the [[Get]] internal method of with argument `"length"`
Deleted: If *relativeStart* is an abrupt completion, return *relativeStart*
Deleted: If *relativeEnd* is an abrupt completion, return *relativeEnd*
Deleted: calling the [[HasProperty]] internal method of ... with argument

c. ReturnIfAbrupt(*kPresent*).
d. If *kPresent* is **true**, then
   i. Let *kValue* be the result of Get(*O*, *Pk*).
   ii. ReturnIfAbrupt(*kValue*).
   iii. Let *status* be the result of CreateOwnDataProperty(*A*, ToString(*n*), *kValue*).
   iv. ReturnIfAbrupt(*status*).
   v. If *status* is **false**, throw a **TypeError** exception.
e. Increase *k* by 1.
f. Increase *n* by 1.

15. Let *putStatus* be the result of Put(*A*, `"length"`, *n*, **true**).
16. ReturnIfAbrupt(*putStatus*).
17. Return *A*.

The `length` property of the `slice` method is **2**.

NOTE 1 The explicit setting of the `length` property of the result Array in step 15 is necessary to ensure that its value is correct in situations where the trailing elements of the result Array are not present.

NOTE 2 The `slice` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `slice` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.11 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*.

Let *obj* be the result of calling ToObject passing the **this** value as the argument.

Let *len* be the result of applying Uint32 to the result of Get(*obj*, `"length"`).

If *comparefn* is not **undefined** and is not a consistent comparison function for the elements of this array (see below), the behaviour of `sort` is implementation-defined.

Let *proto* be the result of calling the [[GetInheritance]] internal method of *obj*. If *proto* is not **null** and there exists an integer *j* such that all of the conditions below are satisfied then the behaviour of `sort` is implementation-defined:

- *obj* is sparse (15.4)
- 0 ≤ *j* < *len*
- The result of HasProperty(*proto*, ToString(*j*)) is **true**.

The behaviour of `sort` is also implementation defined if *obj* is sparse and any of the following conditions are true:

- The result of the predicate IsExtensible(*obj*) is **false**.
- Any array index property of *obj* whose name is a nonnegative integer less than *len* is a data property whose [[Configurable]] attribute is **false**.

The behaviour of `sort` is also implementation defined if any array index property of *obj* whose name is a nonnegative integer less than *len* is an accessor property or is a data property whose [[Writable]] attribute is **false**.

Otherwise, the following steps are taken.

1. Perform an implementation-dependent sequence of calls to the [[Get]] and [[Set]] internal methods of *obj*, to the DeletePropertyOrThrow abstract operation with *obj* as the first argument, and to SortCompare (described below), where the property key argument for each call to [[Get]], [[Set]], or DeletePropertyOrThrow is the

Deleted: calling the [[…et]] internal method of… .
Deleted: If *kValue* is an abrupt completion, return *kValue*
Deleted: Call the [[DefineOwnProperty]] internal method of ... with arguments… ToString(*n*), …,
Deleted: .
Deleted: , and **false**
Comment [AWB9153]: Fix for ES5 bug https://bugs.ecmascript.org/show_bug.cgi?id=417
Deleted: *final*
Deleted: calling the [[Put]] internal method of *A* with arguments `"length"`, *final*, and **true**
Deleted: host
Deleted: 15.4.4
Comment [AWB7154]: TODO: take completion valu… into account.
Deleted: *O*
Deleted: calling the [[Get]] internal method of *obj* with argument `"length"`
Deleted: value …esult of calling the [[Prototype
Deleted: e
Deleted: property
Deleted: calling the [[…asProperty]] internal metho…
Deleted: calling …he predicate [[…sExtensible]]
Deleted: P
Deleted: ,…nd [[Put
Deleted: P
Deleted: and …o the [[…elete]] …ropertyOrThrow inter…
Deleted: P…], [[SetP
Deleted: Put…], or [[Delete]…eleteProperty]

string representation of a nonnegative integer less than *len* and where the arguments for calls to SortCompare are results of previous calls to the [[Get]] internal method. If *obj* is not sparse then DeletePropertyOrThrow must not be called. If any [[Set]] call returns **false** a **TypeError** exception is thrown. If an abrupt completion is returned from any of these operations, it is immediately returned as the value of this function.
2. Return *obj*.

Deleted: P

Deleted: The throw argument to the [[Put]] an [[Delete]] internal methods will be the value tru

Deleted: [[

Deleted:]]

Deleted: P

The returned object must have the following two properties.

- There must be some mathematical permutation π of the nonnegative integers less than *len*, such that for every nonnegative integer *j* less than *len*, if property old[*j*] existed, then new[π(*j*)] is exactly the same value as old[*j*],. But if property old[*j*] did not exist, then new[π(*j*)] does not exist.
- Then for all nonnegative integers *j* and *k*, each less than *len*, if SortCompare(*j*,*k*) < 0 (see SortCompare below), then $\pi(j) < \pi(k)$.

Here the notation old[*j*] is used to refer to the hypothetical result of calling the [[Get]] internal method of *obj* with argument *j* before this function is executed, and the notation new[*j*] to refer to the hypothetical result of calling the [[Get]] internal method of *obj* with argument *j* after this function has been executed.

Deleted: P

Deleted: P

A function *comparefn* is a consistent comparison function for a set of values *S* if all of the requirements below are met for all values *a*, *b*, and *c* (possibly the same value) in the set *S*: The notation $a <_{CF} b$ means *comparefn*(*a*,*b*) < 0; $a =_{CF} b$ means *comparefn*(*a*,*b*) = 0 (of either sign); and $a >_{CF} b$ means *comparefn*(*a*,*b*) > 0.

- Calling *comparefn*(*a*,*b*) always returns the same value *v* when given a specific pair of values *a* and *b* as its two arguments. Furthermore, Type(*v*) is Number, and *v* is not NaN. Note that this implies that exactly one of $a <_{CF} b$, $a =_{CF} b$, and $a >_{CF} b$ will be true for a given pair of *a* and *b*.
- Calling *comparefn*(*a*,*b*) does not modify the **this** object.
- $a =_{CF} a$ (reflexivity)
- If $a =_{CF} b$, then $b =_{CF} a$ (symmetry)
- If $a =_{CF} b$ and $b =_{CF} c$, then $a =_{CF} c$ (transitivity of $=_{CF}$)
- If $a <_{CF} b$ and $b <_{CF} c$, then $a <_{CF} c$ (transitivity of $<_{CF}$)
- If $a >_{CF} b$ and $b >_{CF} c$, then $a >_{CF} c$ (transitivity of $>_{CF}$)

NOTE The above conditions are necessary and sufficient to ensure that *comparefn* divides the set *S* into equivalence classes and that these equivalence classes are totally ordered.

Runtime Semantics: SortCompare Abstract Operation

When the SortCompare abstract operation is called with two arguments *j* and *k*, the following steps are taken:

1. Let *jString* be ToString(*j*).
2. Let *kString* be ToString(*k*).
3. Let *hasj* be the result of HasProperty(*obj*, *jString*).
4. ReturnIfAbrupt(*hasj*).
5. Let *hask* be the result of HasProperty(*obj*, *kString*).
6. ReturnIfAbrupt(*hask*).
7. If *hasj* and *hask* are both **false**, then return **+0**.
8. If *hasj* is **false**, then return 1.
9. If *hask* is **false**, then return –1.
10. Let *x* be the result of Get(*obj*, *jString*).
11. ReturnIfAbrupt(*x*).
12. Let *y* be the result of Get(*obj*, *kString*).
13. ReturnIfAbrupt(*y*).
14. If *x* and *y* are both **undefined**, return **+0**.
15. If *x* is **undefined**, return 1.
16. If *y* is **undefined**, return –1.
17. If the argument *comparefn* is not **undefined**, then
    a. If IsCallable(*comparefn*) is **false**, throw a **TypeError** exception.

Deleted: calling the [[HasProperty]] internal method of

Deleted: with argument

Deleted: calling the [[HasProperty]] internal method of

Deleted: with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

ecma INTERNATIONAL

b. Return the result of calling the [[Call]] internal method of *comparefn* passing **undefined** as *thisArgument* and with a List containing the values of *x* and *y* as the *argumentsList*.

18. Let *xString* be ToString(*x*).
19. ReturnIfAbrupt(*xString*).
20. Let *yString* be ToString(*y*).
21. ReturnIfAbrupt(*yString*).
22. If *xString* < *yString*, return −1.
23. If *xString* > *yString*, return 1.
24. Return **+0**.

NOTE 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 The `sort` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the `sort` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.12 Array.prototype.splice (start, deleteCount [, item1 [, item2 [, …]]])

When the `splice` method is called with two or more arguments *start*, *deleteCount* and (optionally) *item1*, *item2*, etc., the *deleteCount* elements of the array starting at array index *start* are replaced by the arguments *item1*, *item2*, etc. An Array object containing the deleted elements (if any) is returned. The following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
4. Let *lenVal* be the result of Get(*O*, `"length"`).
5. Let *len* be ToUint32(*lenVal*).
6. ReturnIfAbrupt(*len*).
7. Let *relativeStart* be ToInteger(*start*).
8. ReturnIfAbrupt(*relativeStart*).
9. If *relativeStart* is negative, let *actualStart* be max((*len* + *relativeStart*),0); else let *actualStart* be min(*relativeStart*, *len*).
10. Let *actualDeleteCount* be min(max(ToInteger(*deleteCount*),0), *len* – *actualStart*).
11. Let *k* be 0.
12. Repeat, while *k* < *actualDeleteCount*
    a. Let *from* be ToString(*actualStart*+*k*).
    b. Let *fromPresent* be the result of HasProperty(*O*, *from*).
    c. ReturnIfAbrupt(*fromPresent*).
    d. If *fromPresent* is **true**, then
        i. Let *fromValue* be the result of Get(*O*, *from*).
        ii. ReturnIfAbrupt(*fromValue*).
        iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*k*) and Property Descriptor {[[Value]]: *fromValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Increment *k* by 1.
13. Let *putStatus* be the result of Put(*A*, `"length"`, *actualDeleteCount*, **true**).
14. ReturnIfAbrupt(*putStatus*).
15. Let *items* be an internal List whose elements are, in left to right order, the portion of the actual argument list starting with *item1*. The list will be empty if no such items are present.
16. Let *itemCount* be the number of elements in *items*.
17. If *itemCount* < *actualDeleteCount*, then
    a. Let *k* be *actualStart*.
    b. Repeat, while *k* < (*len* – *actualDeleteCount*)
        i. Let *from* be ToString(*k*+*actualDeleteCount*).
        ii. Let *to* be ToString(*k*+*itemCount*).
        iii. Let *fromPresent* be the result of HasProperty(*O*, *from*).
        iv. ReturnIfAbrupt(*fromPresent*).
        v. If *fromPresent* is **true**, then

Deleted: the **this** value
Deleted: arguments
Deleted: host
Deleted: 15.4.4
Deleted: abstraction operation
Deleted: (15.4)
Deleted: a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructo with that name
Deleted: calling the [[Get]] internal method of *O* with argument `"length"`.
Deleted: calling the [[HasProperty]] internal method c
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted:),
Deleted: , and **false**
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: calling the [[HasProperty]] internal metho of
Deleted: with argument

1. Let *fromValue* be the result of Get(*O*, *from*).
2. ReturnIfAbrupt(*fromValue*).
3. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
4. ReturnIfAbrupt(*putStatus*).

vi. Else *fromPresent* is **false**,

1. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
2. ReturnIfAbrupt(*deleteStatus*).

vii. Increase k by 1.

c. Let *k* be *len*.

d. Repeat, while *k* > (*len* – *actualDeleteCount* + *itemCount*)

i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, ToString(*k*–1)).
ii. ReturnIfAbrupt(*deleteStatus*).
iii. Decrease *k* by 1.

18. Else if *itemCount* > *actualDeleteCount*, then

a. Let *k* be (*len* – *actualDeleteCount*).

b. Repeat, while *k* > *actualStart*

i. Let *from* be ToString(*k* + *actualDeleteCount* – 1).
ii. Let *to* be ToString(*k* + *itemCount* – 1)
iii. Let *fromPresent* be the result of HasProperty(*O*, *from*).
iv. ReturnIfAbrupt(*fromPresent*).
v. If *fromPresent* is **true**, then

1. Let *fromValue* be the result of Get(*O*, *from*).
2. ReturnIfAbrupt(*fromValue*).
3. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
4. ReturnIfAbrupt(*putStatus*).

vi. Else *fromPresent* is **false**,

1. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
2. ReturnIfAbrupt(*deleteStatus*).

vii. Decrease *k* by 1.

19. Let *k* be *actualStart*.

20. Repeat, while *items* is not empty

a. Remove the first element from *items* and let *E* be the value of that element.
b. Let *putStatus* be the result of Put(*O*, ToString(*k*), *E*, **true**).
c. ReturnIfAbrupt(*putStatus*).
d. Increase *k* by 1.

21. Let *putStatus* be the result of Put(*O*, `"length"`, *len* – *actualDeleteCount* + *itemCount*, **true**).
22. ReturnIfAbrupt(*putStatus*).
23. Return *A*.

The `length` property of the `splice` method is **2**.

NOTE 1 The explicit setting of the `length` property of the result Array in step 13 is necessary to ensure that its value is correct in situations where its trailing elements are not present.

NOTE 2 The `splice` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `splice` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.13 Array.prototype.unshift ([item1 [, item2 [, …]]])

The arguments are prepended to the start of the array, such that their order within the array is the same as the order in which they appear in the argument list.

When the `unshift` method is called with zero or more arguments *item1*, *item2*, etc., the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenVal*).

Deleted: calling the [[…et]] internal
Deleted: Call
Deleted: calling the [[…ut]] internal
Deleted: ,
Deleted: Call
Deleted: calling the [[…elete]] interna
Deleted: calling Call the
Deleted: calling the [[HasProperty]] internal method of … with argument
Deleted: calling the [[…et]] internal
Deleted: calling Call the [[…ut]] inte
Deleted: ,
Deleted: calling Call the
Deleted: calling Call the [[…ut]] internal
Deleted: calling Call the [[…ut]] internal meth
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method o with argument `"length"`.

5. ReturnIfAbrupt(*len*).
6. Let *argCount* be the number of actual arguments.
7. Let *k* be *len*.
8. Repeat, while *k* > 0,
    a. Let *from* be ToString(*k*–1).
    b. Let *to* be ToString(*k*+*argCount* –1).
    c. Let *fromPresent* be the result of HasProperty(*O*, *from*).
    d. ReturnIfAbrupt(*fromPresent*).
    e. If *fromPresent* is **true**, then
        i. Let *fromValue* be the result of Get(*O*, *from*).
        ii. ReturnIfAbrupt(*fromValue*).
        iii. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
        iv. ReturnIfAbrupt(*putStatus*).
    f. Else *fromPresent* is **false**,
        i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
        ii. ReturnIfAbrupt(*deleteStatus*).
    g. Decrease *k* by 1.
9. Let *j* be 0.
10. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
11. Repeat, while *items* is not empty
    a. Remove the first element from *items* and let *E* be the value of that element.
    b. Let *putStatus* be the result of Put(*O*, ToString(*j*), *E*, **true**).
    c. ReturnIfAbrupt(*putStatus*).
    d. Increase *j* by 1.
12. Let *putStatus* be the result of Put(*O*, **"length"**, *len*+*argCount*, **true**).
13. ReturnIfAbrupt(*putStatus*).
14. Return *len*+*argCount*.

The **length** property of the **unshift** method is **1**.

NOTE The **unshift** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **unshift** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.14 Array.prototype.indexOf (searchElement [, fromIndex])

indexOf compares *searchElement* to the elements of the array, in ascending order, using the Strict Equality Comparison algorithm (11.9.1), and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, the whole array will be searched.

When the **indexOf** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **"length"**)
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If *len* is 0, return -1.
7. If argument *fromIndex* was passed let *n* be ToInteger(*fromIndex*); else let *n* be 0.
8. ReturnIfAbrupt(*n*).
9. If *n* ≥ *len*, return -1.
10. If *n* ≥ 0, then
    a. Let *k* be *n*.

Deleted: calling the [[HasProperty]] internal method o
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: calling Call the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: ,
Deleted: calling Call the [[
Deleted:]] internal method of
Deleted: with arguments
Deleted: , and **true**
Deleted: calling Call the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: calling Call the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: host
Deleted: 15.4.4
Deleted: internal
Deleted: Algorithm
Deleted: 6
Deleted: calling the [[Get]] internal method of *O* with the argument **"length"**.
Deleted: If *n* is an abrupt completion, return *n*.

11. Else, *n*<0,
    a. Let *k* be *len* - abs(*n*).
    b. If *k* < 0, then let *k* be 0.
12. Repeat, while *k*<*len*
    a. Let *kPresent* be the result of HasProperty(*O*, ToString(*k*)).
    b. ReturnIfAbrupt(*kPresent*).
    c. If *kPresent* is **true**, then
        i. Let *elementK* be the result of Get(*O*, ToString(*k*)).
        ii. ReturnIfAbrupt(*elementK*).
        iii. Let *same* be the result of performing Strict Equality Comparison *searchElement* === *elementK*.
        iv. If *same* is **true**, return *k*.
    d. Increase *k* by 1.
13. Return -1.

The **`length`** property of the **`indexOf`** method is **1**.

NOTE The **`indexOf`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`indexOf`** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.15 Array.prototype.lastIndexOf (searchElement [, fromIndex])

`lastIndexOf` compares *searchElement* to the elements of the array in descending order using the Strict Equality Comparison algorithm (11.9.1), and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to the array's length minus one (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, -1 is returned.

When the **`lastIndexOf`** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **`"length"`**).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If *len* is 0, return -1.
7. If argument *fromIndex* was passed let *n* be ToInteger(*fromIndex*); else let *n* be *len*-1.
8. ReturnIfAbrupt(*n*).
9. If *n* ≥ 0, then let *k* be min(*n*, *len* – 1).
10. Else, *n* < 0,
    a. Let *k* be *len* - abs(*n*).
11. Repeat, while *k*≥ 0
    a. Let *kPresent* be the result of HasProperty(*O*, ToString(*k*)).
    b. ReturnIfAbrupt(*kPresent*).
    c. If *kPresent* is **true**, then
        i. Let *elementK* be the result of Get(*O*, ToString(*k*)).
        ii. ReturnIfAbrupt(*elementK*).
        iii. Let *same* be the result of performing Strict Equality Comparison *searchElement* === *elementK*.
        iv. If *same* is **true**, return *k*.
    d. Decrease *k* by 1.
12. Return -1.

The **`length`** property of the **`lastIndexOf`** method is **1**.

Deleted: ,
Deleted: is less than
Deleted: calling the [[HasProperty]] internal method of
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with the argument
Deleted: If *elementK* is an abrupt completion, return *elementK*.
Deleted: applying
Deleted: the
Deleted: Algorithm
Deleted: to
Deleted: and
Deleted: host
Deleted: 15.4.4
Deleted: internal
Deleted: Algorithm
Deleted: 6
Deleted: calling the [[Get]] internal method of with the argument **`"length"`**.
Deleted: If *n* is an abrupt completion, return *n*
Deleted: ,
Deleted: calling the [[HasProperty]] internal method of
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with the argument
Deleted: If *elementK* is an abrupt completion, return *elementK*.
Deleted: applying
Deleted: the
Deleted: Algorithm
Deleted: Algorithm
Deleted: to
Deleted: and

NOTE The `lastIndexOf` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `lastIndexOf` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.16 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 the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*
   a. Let *Pk* be ToString(*k*).
   b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
   c. If *kPresent* is **true**, then
      i. Let *kValue* be the result of Get(*O*, *Pk*).
      ii. ReturnIfAbrupt(*kValue*).
      iii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
      iv. ReturnIfAbrupt(*testResult*).
      v. If ToBoolean(*testResult)* is **false**, return **false**.
   d. Increase *k* by 1.
10. Return **true**.

The `length` property of the `every` method is **1**.

NOTE The `every` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `every` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.17 Array.prototype.some (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**. `some` calls *callbackfn* once for each element present in the array, in ascending

Deleted: host

Deleted: 15.4.4

Deleted: calling the [[Get]] internal method of *O* with the argument `"length"`.

Deleted: calling the [[HasProperty]] internal method o

Deleted: with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: If *kValue* is an abrupt completion, return *kValue*.

Deleted: the **this** value

Deleted: argument

Deleted: list

Deleted: If *testResult* is an abrupt completion, retur *testResult*.

Deleted: host

Deleted: 15.4.4

order, until it finds one where *callbackfn* returns **true**. If such an element is found, **some** immediately returns **true**. Otherwise, **some** returns **false**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

some does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **some** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **some** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time that **some** visits them; elements that are deleted after the call to **some** begins and before being visited are not visited. **some** acts like the "exists" quantifier in mathematics. In particular, for an empty array, it returns **false**.

When the **some** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*
   a. Let *Pk* be ToString(*k*).
   b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
   c. ReturnIfAbrupt(*kPresent*).
   d. If *kPresent* is **true**, then
      i. Let *kValue* be the result of Get(*O*, *Pk*).
      ii. ReturnIfAbrupt(*kValue*).
      iii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
      iv. ReturnIfAbrupt(*testResult*).
      v. If ToBoolean(*testResult*) is **true**, return **true**.
   e. Increase *k* by 1.
10. Return **false**.

The **length** property of the **some** method is **1**.

NOTE The **some** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **some** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.18 Array.prototype.forEach (callbackfn [, thisArg])

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.

Deleted: calling the [[Get]] internal method of with the argument **"length"**

Deleted: calling the [[HasProperty]] internal method of

Deleted: with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: If *kValue* is an abrupt completion, return *kValue*.

Deleted: the **this** value

Deleted: argument

Deleted: list

Deleted: If *testResult* is an abrupt completion, return *testResult*.

Deleted: host

Deleted: 15.4.4

forEach does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **forEach** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **forEach** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to callback will be the value at the time **forEach** visits them; elements that are deleted after the call to **forEach** begins and before being visited are not visited.

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*funcResult*).
    e. Increase *k* by 1.
10. Return **undefined**.

The **length** property of the **forEach** method is **1**.

NOTE The **forEach** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **forEach** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.19 Array.prototype.map (callbackfn [, thisArg])

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:

Deleted: calling the [[Get]] internal method of *O* with the argument **"length"**.

Deleted: calling the [[HasProperty]] internal method o with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: If *kValue* is an abrupt completion, return *kValue*.

Deleted: Call

Deleted: the **this** value

Deleted: argument

Deleted: list

Deleted: If *funcResult* is an abrupt completion, retu *funcResult*.

Deleted: host

Deleted: 15.4.4

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *A* be the result of the abstract operation ArrayCreate with argument *len*.
9. Let *k* be 0.
10. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *mappedValue* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*mappedValue*).
        v. Call the [[DefineOwnProperty]] internal method of *A* with arguments *Pk* and Property Descriptor {[[Value]]: *mappedValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Increase *k* by 1.
11. Return *A*.

The **length** property of the **map** method is **1**.

NOTE The **map** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **map** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.20 Array.prototype.filter (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**. **filter** calls *callbackfn* once for each element in the array, in ascending order, and constructs a new array of all the values for which *callbackfn* returns **true**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

filter does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **filter** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **filter** begins will not be visited by *callbackfn*. If existing elements of the array are changed their value as passed to *callbackfn* will be the value at the time **filter** visits them; elements that are deleted after the call to **filter** begins and before being visited are not visited.

When the **filter** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.

Deleted: calling the [[Get]] internal method of with the argument **"length"**.

Deleted: abstraction operation

Deleted: (15.4)

Deleted: a new array created as if by the expression **new Array(*len*)** where **Array** is standard built-in constructor with that name and is the value of

Deleted: calling the [[HasProperty]] interna method of *O* with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: the **this** value

Deleted: argument

Deleted: list

Deleted: ,

Deleted: , and **false**

Deleted: host

Deleted: 15.4.4

Deleted: calling the [[Get]] internal method of with the argument **"length"**

7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
9. Let *k* be 0.
10. Let *to* be 0.
11. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *selected* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*selected*).
        v. If ToBoolean(*selected*) is **true**, then
            1. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*to*) and Property Descriptor {[[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            2. Increase *to* by 1.
    e. Increase *k* by 1.
12. Return *A*.

The `length` property of the `filter` method is **1**.

NOTE The `filter` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `filter` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.21 Array.prototype.reduce (callbackfn [, initialValue])

callbackfn should be a function that takes four arguments. `reduce` calls the callback, as a function, once for each element present in the array, in ascending order.

callbackfn is called with four arguments: the *previousValue* (or value from the previous call to *callbackfn*), the *currentValue* (value of the current element), the *currentIndex*, and the object being traversed. The first time that callback is called, the *previousValue* and *currentValue* can be one of two values. If an *initialValue* was provided in the call to `reduce`, then *previousValue* will be equal to *initialValue* and *currentValue* will be equal to the first value in the array. If no *initialValue* was provided, then *previousValue* will be equal to the first value in the array and *currentValue* will be equal to the second. It is a **TypeError** if the array contains no elements and *initialValue* is not provided.

`reduce` does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by `reduce` is set before the first call to *callbackfn*. Elements that are appended to the array after the call to `reduce` begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time `reduce` visits them; elements that are deleted after the call to `reduce` begins and before being visited are not visited.

When the `reduce` method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
8. Let *k* be 0.
9. If *initialValue* is present, then

Deleted: abstraction operation
Deleted: (15.4)
Deleted: a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructo with that name
Deleted: calling the [[HasProperty]] internal method o with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *kValue* is an abrupt completion, return *kValue*
Deleted: the **this** value
Deleted: argument
Deleted: list
Deleted: If *selected* is an abrupt completion, return *selected*.
Deleted:),
Deleted: , and **false**
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method of *O* with th argument `"length"`

a. Set *accumulator* to *initialValue*.

10. 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 the result of HasProperty(*O*, *Pk*).
        iii. ReturnIfAbrupt(*kPresent*).
        iv. If *kPresent* is **true**, then
            1. Let *accumulator* be the result of Get(*O*, *Pk*).
            2. ReturnIfAbrupt(*accumulator*).
        v. Increase *k* by 1.
    c. If *kPresent* is **false**, throw a **TypeError** exception.
11. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as *thisArgument* and a List containing *accumulator*, *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*accumulator*).
    e. Increase *k* by 1.
12. Return *accumulator*.

The `length` property of the `reduce` method is **1**.

NOTE The `reduce` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `reduce` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.22 Array.prototype.reduceRight (callbackfn [, initialValue])

callbackfn should be a function that takes four arguments. `reduceRight` calls the callback, as a function, once for each element present in the array, in descending order.

callbackfn is called with four arguments: the *previousValue* (or value from the previous call to *callbackfn*), the *currentValue* (value of the current element), the *currentIndex*, and the object being traversed. The first time the function is called, the *previousValue* and *currentValue* can be one of two values. If an *initialValue* was provided in the call to `reduceRight`, then *previousValue* will be equal to *initialValue* and *currentValue* will be equal to the last value in the array. If no *initialValue* was provided, then *previousValue* will be equal to the last value in the array and *currentValue* will be equal to the second-to-last value. It is a **TypeError** if the array contains no elements and *initialValue* is not provided.

`reduceRight` does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by `reduceRight` is set before the first call to *callbackfn*. Elements that are appended to the array after the call to `reduceRight` begins will not be visited by *callbackfn*. If existing elements of the array are changed by *callbackfn*, their value as passed to *callbackfn* will be the value at the time `reduceRight` visits them; elements that are deleted after the call to `reduceRight` begins and before being visited are not visited.

When the `reduceRight` method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenValue*).

Deleted: ,
Deleted: calling the [[HasProperty]] internal method of *O* with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *accumulator* is an abrupt completion, return *accumulator*.
Deleted: calling the [[HasProperty]] interna method of *O* with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *kValue* is an abrupt completio return *kValue*
Deleted: the **this** value
Deleted: argument
Deleted: list
Deleted: If *accumulator* is an abrupt completion, return *accumulator*
Deleted: host
Deleted: 15.4.4
Deleted: calling the [[Get]] internal method of with the argument `"length"`

5. ReturnIfAbrupt(*len*).
6. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
7. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
8. Let *k* be *len*-1.
9. If *initialValue* is present, then
   a. Set *accumulator* to *initialValue*.
10. Else *initialValue* is not present,
    a. Let *kPresent* be **false**.
    b. Repeat, while *kPresent* is **false** and $k \geq 0$
       i. Let *Pk* be ToString(*k*).
       ii. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
       iii. ReturnIfAbrupt(*kPresent*).
       iv. If *kPresent* is **true**, then
           1. Let *accumulator* be the result of Get(*O*, *Pk*).
           2. ReturnIfAbrupt(*accumulator*).
       v. Decrease *k* by 1.
    c. If *kPresent* is **false**, throw a **TypeError** exception.
11. Repeat, while $k \geq 0$
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
       i. Let *kValue* be the result of Get(*O*, *Pk*).
       ii. ReturnIfAbrupt(*kValue*).
       iii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as *thisArgument* and a List containing *accumulator*, *kValue*, *k*, and *O* as *argumentsList*.
       iv. ReturnIfAbrupt(*accumulator*).
    e. Decrease *k* by 1.
12. Return *accumulator*.

The `length` property of the `reduceRight` method is **1**.

NOTE The `reduceRight` function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `reduceRight` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.23 Array.prototype.find (predicate , thisArg = undefined)

predicate should be a function that accepts three arguments and returns a value that is coercible to the Boolean value **true** or **false**. `find` calls *predicate* once for each element present in the array, in ascending order, until it finds one where *predicate* returns **true**. If such an element is found, `find` immediately returns that element value. Otherwise, `find` returns **undefined**. *predicate* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *predicate*. If it is not provided, **undefined** is used instead.

predicate is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`find` does not directly mutate the object on which it is called but the object may be mutated by the calls to *predicate*.

The range of elements processed by `find` is set before the first call to *callbackfn*. Elements that are appended to the array after the call to `find` begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *predicate* will be the value at the time that `find` visits them; elements that are deleted after the call to `find` begins and before being visited are not visited.

When the `find` method is called with one or two arguments, the following steps are taken:

Deleted: ,
Deleted: calling the [[HasProperty]] internal method of *O* with argument
Deleted: of calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *accumulator* is an abrupt completion, return *accumulator*
Deleted: calling the [[HasProperty]] internal method of with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: If *kValue* is an abrupt completion, return *kValue*
Deleted: the **this** value
Deleted: argument
Deleted: list
Deleted: If *accumulator* is an abrupt completion, return *accumulator*
Deleted: host

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, `"length"`).
4. Let *len* be ToInteger(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*predicate*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *testResult* be the result of calling the [[Call]] internal method of *predicate* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*testResult*).
        v. If ToBoolean(*testResult*) is **true**, return *kValue*.
    e. Increase *k* by 1.
10. Return **undefined**.

The **`length`** property of the **`find`** method is **1**.

NOTE The **`find`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`find`** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.24 Array.prototype.findIndex (predicate , thisArg = undefined)

predicate should be a function that accepts three arguments and returns a value that is coercible to the Boolean value **true** or **false**. **`findIndex`** calls *predicate* once for each element present in the array, in ascending order, until it finds one where *predicate* returns **true**. If such an element is found, **`findIndex`** immediately returns the index of that element value. Otherwise, **`findIndex`** returns -1. *predicate* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *predicate*. If it is not provided, **undefined** is used instead.

predicate is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`findIndex` does not directly mutate the object on which it is called but the object may be mutated by the calls to *predicate*.

The range of elements processed by **`findIndex`** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **`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 **`findIndex`** visits them; elements that are deleted after the call to **`findIndex`** begins and before being visited are not visited.

When the **`findIndex`** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenValue* be the result of Get(*O*, `"length"`).
4. Let *len* be ToInteger(*lenValue*).
5. ReturnIfAbrupt(*len*).
6. If IsCallable(*predicate*) is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.

9. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *testResult* be the result of calling the [[Call]] internal method of *predicate* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*testResult*).
        v. If ToBoolean(*testResult*) is **true**, return *k*.
    e. Increase *k* by 1.
10. Return -1.

The **`length`** property of the **`findIndex`** method is **1**.

NOTE The **`findIndex`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`findIndex`** function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

15.4.3.25 Array.prototype.entries ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and **`"key+value"`**.

15.4.3.26 Array.prototype.keys ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and **`"key"`**.

15.4.3.27 Array.prototype.values ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and **`"value"`**.

15.4.3.28 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.

15.4.4 Properties of Array Instances

Array instances are exotic Array objects and have the internal methods specified for such objects. Array instances inherit properties from the Array prototype object. Array instances also have a [[ArrayInitialisationState]] internal data property.

Array instances have a **`length`** property, and a set of enumerable properties with array index names.

Comment [AWB11155]: Need to decide whether to allow an argument that requests sparse iteration

Deleted: 15.4.4

Deleted: items

Comment [AWB11156]: Need to decide whether to allow an argument that requests sparse iteration

Deleted: 15.4.4

Comment [AWB11157]: Need to decide whether to allow an argument that requests sparse iteration

Deleted: 15.4.4

Comment [AWB10158]: This is how we identify a property whose key is a built-in Symbol

Deleted: 15.4.4

Deleted: .

Comment [AWB10159]: Do we really want to do this sort of method sharing. It has a bad smell.

Deleted: items

Deleted: that

Deleted: and

Deleted: their [[Class]]

Deleted: the

Deleted: IsArray

Deleted: NativeBrand

Deleted: z

Deleted: BuiltinBrand

Deleted: with value NativeArrayBuiltinArray

Deleted: value is **`"Array"`**

Deleted: Array instances also have the following properties.

15.4.4.1 length

The **length** property of this Array object is a data property whose value is always numerically greater than the name of every deletable property whose name is an array index.

The **length** property initially has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE Attempting to set the length property of an Array object to a value that is numerically less than or equal to the largest numeric property name of an existing array indexed non-deletable property of the array will result in the length being set to a numeric value that is one greater than that largest numeric property name. See 8.4.2.1.

15.4.5 Array Iterator Object Structure

An Array Iterator is an object, with the structure defined below, that represent 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.

15.4.5.1 CreateArrayIterator Abstract Operation

Several methods of Array objects return Iterator objects. The abstract operation CreateArrayIterator with arguments *array* and *kind* is used to create such iterator objects. It performs the following steps:

1. Let *O* be the result of calling ToObject(*array*).
2. ReturnIfAbrupt(*O*).
3. Let *iterator* be the result of the abstract operation ObjectCreate with the intrinsic object %ArrayIteratorPrototype% as its argument.
4. Add a [[IteratedObject]] internal data property to *iterator* with value *O*.
5. Add a [[ArrayIteratorNextIndex]] internal data property to *iterator* with value 0.
6. Add a [[ArrayIterationKind]] internal data property of *iterator* with value *kind*.
7. Return *iterator*.

15.4.5.2 The Array Iterator Prototype

All Array Iterator Objects inherit properties from a common Array Iterator Prototype object. The [[Prototype]] internal data property of the Array Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Array Iterator Prototype as the following properties:

15.4.5.2.1 ArrayIterator.prototype.constructor

15.4.5.2.2 ArrayIterator.prototype.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 properties of a Array Iterator Instance (15.4.5.1.2), throw a **TypeError** exception.
4. Let *a* be the value of the [[IteratedObject]] internal data property of *O*.
5. Let *index* be the value of the [[ArrayIteratorNextIndex]] internal data property of *O*.
6. Let *itemKind* be the value of the [[ArrayIterationKind]] internal data property of *O*.
7. Let *lenValue* be the result of Get(*a*, **"length"**).
8. Let *len* be ToUint32(*lenValue*).
9. ReturnIfAbrupt(*len*).
10. If *itemKind* contains the substring **"sparse"**, then
    a. Let *found* be **false**.
    b. Repeat, while *found* is **false** and *index* < *len*
        i. Let *elementKey* be ToString(*index*).
        ii. Let *found* be the result of HasProperty(*a*, *elementKey*).
        iii. ReturnIfAbrupt(*found*).
        iv. If *found* is **false**, then
            1. Increase *index* by 1.

Deleted: <#>15.4.5.1 [[DefineOwnProperty]] (P, Desc, Throw)¶
Array objects use a variation of the [[DefineOwnProperty]] internal method used for other native ECMAScript objects (8.12.9).¶
Assume *A* is an Array object, *Desc* is a Property Descriptor, and *Throw* is a Boolean flag.¶
In the following algorithm, the term "Reject" means "If *Throw* is **true**, then throw a **TypeError** exception, otherwise return **false**."¶
When the [[DefineOwnProperty]] internal method of *A* is called with property *P*, Property Descriptor *Desc*, and Boolean flag *Throw*, the following steps are taken:¶
<#>ReturnIfAbrupt(*A*).¶
<#>Let *oldLenDesc* be the result of calling the [[GetOwnProperty]] internal method of *A* passing **"length"** as the argument. The result will never be **undefined** or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.¶
<#>Let *oldLen* be *oldLenDesc*.[[Value]].¶
<#>If *P* is **"length"**, then¶
<#>If the [[Value]] field of *Desc* is absent, then
<#>Return the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing **"length"**, *Desc*, and *Throw* as arguments.¶
<#>Let *newLenDesc* be a copy of *Desc*.¶
<#>Let *newLen* be ToUint32(*Desc*.[[Value]]).
<#>If *newLen* is not equal to ToNumber(*Desc*.[[Value]]), throw a **RangeError** exception.
<#>Set *newLenDesc*.[[Value]] to *newLen*.¶
<#>If *newLen* ≥*oldLen*, then¶
<#>Return the result of calling the default [[DefineOwnProperty]] internal method

Deleted: 15

Deleted: 5

Deleted: 15.4.6

Deleted: 15.4.6

Deleted: interator

Deleted: *itr*

Deleted: *itr*

Deleted: *itr*

Deleted: *itr*

Deleted: *itr*

Deleted: 15.4.6

Comment [AWB10160]: TODO: need to decide what to use for a constructor for these sort of objects.

Deleted: 15.4.6

Deleted: 15.4.6

Deleted: 15.4.6

Deleted: calling the [[Get]] internal method of with the argument **"length"**

Deleted: calling the [[HasProperty]] internal method of *a* with argument

Comment [AWB15161]: We don't current have a public API for requesting a sparse

11. If *index* ≥ *len*, then
    a. Set the value of the [[ArrayIteratorNextIndex]] internal data property of *O* to +∞.
    b. Return CreateItrResultObject(**undefined**, **true**).
12. Let *elementKey* be ToString(*index*).
13. Set the value of the [[ArrayIteratorNextIndex]] internal data property of *O* to *index*+1.
14. If *itemKind* contains the substring **"value"**, then
    a. Let *elementValue* be the result of Get(*a*, *elementKey*).
    b. ReturnIfAbrupt(*elementValue*).
15. If *itemKind* contains the substring **"key+value"**, then
    a. Let *result* be the result of the abstract operation ArrayCreate with argument 2.
    b. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
    c. Call the [[DefineOwnProperty]] internal method of *result* with arguments **"0"** and Property Descriptor {[[Value]]: *elementKey*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    d. Call the [[DefineOwnProperty]] internal method of *result* with arguments **"1"** and Property Descriptor {[[Value]]: *elementValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Return CreateItrResultObject(*result*, **false**).
16. Else If *itemKind* contains the substring **"key"** then, return CreateItrResultObject(*elementKey*, **false**).
17. Assert: *itemKind* contains the substring **"value"**.
18. Return CreateItrResultObject(*elementValue*, **false**).

15.4.5.2.3 *ArrayIterator*.prototype.@@iterator ( )

The following steps are taken:

1. Return the **this** value.

15.4.5.2.4 *ArrayIterator*.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **"Array Iterator"**.

15.4.5.3 Properties of Array Iterator Instances

Array Iterator instances inherit properties from the Array Iterator prototype (the intrinsic, %ArrayIteratorPrototype%.) Array Iterator instances are initially created with the internal properties listed in Table 33.

Table 33 — Internal Data Properties of Array Iterator Instances

Internal Data Property Name	Description
[[IteratedObject]]	The object whose array elements are being iterated.
[[ArrayIteratorNextIndex]]	The integer index of the next array index to be examined by this iteration.
[[ArrayIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: **"key"**, **"value"**, **"key+value"**, **"sparse:key"**, **"sparse:value"**, **"sparse:key+value"**.

15.5 String Objects

15.5.1 The String Constructor

The String constructor is the %String% instrinsic object and the initial value of the **String** property of the global object. When **String** is called as a function rather than as a constructor, it performs a type conversion.

Deleted: Infinity
Deleted: Completion
Deleted: {[[type]]: throw, [[value]]: %StopIteration% [[target]]: empty}
Deleted: calling the [[
Deleted:]] internal method of
Deleted: with argument
Deleted: ,
Comment [AWB11162]: This is a string. Should be return the numeric value of *index* instead?
Deleted: , and **false**
Deleted: ,
Deleted: , and **false**
Deleted: *result*
Comment [AWB11163]: This is a string. Should we return the numeric value of *index* instead?
Deleted: Else
Deleted: is
Comment [AWB10164]: This is how we identify a property whose key is a built-in Symbol
Deleted: 15.4.6
Deleted: 15.4.6
Deleted: 15.4.6
Deleted: Called as a Function

However, if the **this** value value passed in the call is an Object with an uninitialised [[StringData]] internal data property, it initialises the **this** value using the argument value. This permits `String` to be used both to perform type conversion and to perform constructor instance initialization.

The `String` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified `String` behaviour must include a `super` call to the `String` constructor to initialise the [[StringData]] state of subclass instances.

15.5.1.1 **String ([value])**

Returns a String value (not a String object) computed by ToString(*value*). If *value* is not supplied, the empty String `""` is returned.

When `String` is called with argument *value*, the following steps are taken:

1. Let *O* be the **this** value.
2. If no arguments were passed to this function invocation, then let *s* be `""`.
3. Else, let *s* be ToString(*value*).
4. ReturnIfAbrupt(*s*).
5. If Type(*O*) is Object and *O* has a [[StringData]] internal data property and the value of [[StringData]] is **undefined**, then
   a. Let *length* be the number of code unit elements in *s*.
   b. Let *status* be the result of DefinePropertyOrThrow(*O*, `"length"`, Property Descriptor{[[Value]]: *length*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
   c. ReturnIfAbrupt(*status*).
   d. Set the value of *O's* [[StringData]] internal data property to *s*.
   e. Return *O*.
6. Return *s*.

15.5.1.2 **new String (... argumentsList)**

`String` called as part of a new expression , it initialises a newly created exotic String object:

1. Let *F* be the `String` function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If `String` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.5.2 Properties of the String Constructor

The value of the [[Prototype]] internal data property of the String constructor is the standard built-in Function prototype object (15.3.3).

Besides the `length` property (whose value is **1**), the String constructor has the following properties:

15.5.2.1 **String.prototype**

The initial value of `String.prototype` is the standard built-in String prototype object (15.5.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.5.2.2 **String.fromCharCode (...codeUnits)**

The `String.fromCharCode` function may be called with a variable number of arguments which form the rest parameter *codeUnits*. The following steps are taken:

Deleted: z

Deleted: ize

Deleted: `String` may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

Deleted: *value* is **undefined**

Deleted: <#>15.5.2 The String Constructor¶
When `String` is called as part of a `new` expression, it is a constructor: it initialises the newly created exotic String object.¶
15.5.2.1

Deleted:

Deleted: [

Deleted: value

Deleted:]

Deleted: with optional argument *value* perfor the following steps

Deleted: be the be the

Deleted: a

Deleted: The [[Prototype]] internal data property of the newly constructed object is se to the standard built-in String prototype objec that is the initial value of `String.prototyp` (15.5.3.1).¶
The [[Class]] internal property of the newly constructed object is set to `"String"` has the [[IsStringNativeBrand]] internal property with value StringWrapper.¶
The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
The [[PrimitiveValueStringData]] internal data property of the newly constructed object is se to ToString(*value*), or to the empty String if *val* is not supplied.¶

Deleted: 15.3.4

Deleted: 15.3.3

Deleted: the internal properties and

Comment [AWB9165]: Should we provide fromCodeUnit as an alias for this property an lable fromCharCode as obsolete.

Deleted: [

Deleted: ...

Deleted: char0 [, char1 [, ...]]]

Deleted: Returns a String value containing a many characters as the number of argument Each argument specifies one character of the resulting String, with the first argument specifying the first character, and so on, from left to right. An argument is converted to a character by applying the operation ToUint16 (9.7) and regarding the resulting 16-bit intege as the code unit value of a character. If no arguments are supplied, the result is the emp String.¶

1. Assert: *codeUnits* is a well-formed rest parameter object.
2. Let *length* be the result of Get(*codeUnits*, `"length"`).
3. Let *elements* be a new List.
4. Let *nextIndex* be 0.
5. Repeat while *nextIndex* < *length*
    a. Let *next* be the result of Get(*codeUnits*, ToString(*nextIndex*)).
    b. Let *nextCU* be ToUint16(*next*).
    c. ReturnIfAbrupt(*nextCU*).
    d. Append *nextCU* to the end of *elements*.
    e. Let *nextIndex* be *nextIndex* + 1.
6. Return the String value whose elements are, in order, the elements in the List *elements*. If *length* is 0, the empty string is returned.

The `length` property of the `fromCharCode` function is **1**.

15.5.2.3 String.fromCodePoint (...codePoints)

The `String.fromCodePoint` function may be called with a variable number of arguments which form the rest parameter *codePoints*. The following steps are taken:

1. Assert: *codePoints* is a well-formed rest parameter object.
2. Let *length* be the result of Get(*codePoints*, `"length"`).
3. Let *elements* be a new List.
4. Let *nextIndex* be 0.
5. Repeat while *nextIndex* < *length*
    a. Let *next* be the result of Get(*codePoints*, ToString(*nextIndex*)).
    b. Let *nextCP* be ToNumber(*next*).
    c. ReturnIfAbrupt(*nextCP*).
    d. If SameValue(*nextCP*, ToInteger(*nextCP*)) is **false**,then throw a **RangeError** exception.
    e. If *nextCP* < 0 or *nextCP* > 0x10FFFF, then throw a **RangeError** exception.
    f. Append the elements of the UTF-16 Encoding (clause 6) of *nextCP* to the end of *elements*.
    g. Let *nextIndex* be *nextIndex* + 1.
6. Return the String value whose elements are, in order, the elements in the List *elements*. If *length* is 0, the empty string is returned.

The `length` property of the `fromCodePoint` function is **0**.

15.5.2.4 String.raw (callSite, ...substitutions)

The `String.raw` function may be called with a variable number of arguments. The first argument is *callSite* and the remainder of the arguments form the rest parameter *substitutions*. The following steps are taken:

1. Assert: *substitutions* is a well-formed rest parameter object.
2. Let *cooked* be ToObject(*callSite*).
3. ReturnIfAbrupt(*cooked*).
4. Let *rawValue* be the result of Get(*cooked*, `"raw"`).
5. Let *raw* be ToObject(*rawValue*).
6. ReturnIfAbrupt(*raw*).
7. Let *len* be the result of Get(*raw*, `"length"`).
8. Let *literalSegments* be ToUint(*len*).
9. ReturnIfAbrupt(*literalSegments*).
10. If *literalSegments* = 0, then return the empty string.
11. Let *stringElements* be a new List.
12. Let *nextIndex* be 0.
13. Repeat
    a. Let *nextKey* be ToString(*nextIndex*).
    b. Let *next* be the result of Get(*raw*, *nextKey*).
    c. Let *nextSeg* be ToString(*next*).
    d. ReturnIfAbrupt(*nextSeg*).

Deleted: ing
Deleted: calling the [[Get]] internal method of *codeUnits* with argument `"length"`
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: s
Deleted: ...
Deleted: `CharCode`
Deleted: *Units*
Deleted: ing
Deleted: calling the [[Get]] internal method of *codeUnitsPoints* with argument `"length"`
Deleted: calling the [[
Deleted:]] internal method of
Deleted: with argument
Deleted: v
Deleted: V
Deleted: s
Comment [AWB9166]: Note this follows the ES6 convention for rest parameters rather than the precede established by fromCharCode
Deleted: ...
Deleted: ing
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: ing
Deleted: calling the [[Get]] internal method of *raw* with argument `"length"`
Comment [AWB10167]: TODO: this is something w probably need if we are going to eliminate array usage ToUint32
Deleted: while *nextIndex* < *literalSegments*
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument

e. Append in order the code unit elements of *nextSeg* to the end of *stringElements*.
f. If *nextIndex* + 1 = *literalSegments*, then
   i. Return the string value whose elements are, in order, the elements in the List *stringElements*. If *length* is 0, the empty string is returned.
g. Let *next* be the result of Get(*substitutions*, *nextKey*).
h. Let *nextSub* be ToString(*next*).
i. ReturnIfAbrupt(*nextSub*).
j. Append in order the code unit elements of *nextSub* to the end of *stringElements*.
k. Let *nextIndex* be *nextIndex* + 1.

The **`length`** property of the **`raw`** function is **1**.

NOTE String.raw is intended for use as a tag function of a Tagged Template String (11.2.6). When called as such the first argument will be a well formed template call site object and the rest parameter will contain the substitution values.

15.5.2.5 String[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *proto* be the result of GetPrototypeFromConstructor(*F*, **`"%StringPrototype%"`**).
3. ReturnIfAbrupt(*proto*).
4. Let *obj* be the result of calling StringCreate (*proto*).
5. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[StringData]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the String constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.5.3 Properties of the String Prototype Object

The String prototype object is itself an ordinary object. It is not a String instance and does not have a [[StringData]] internal data property.

The value of the [[Prototype]] internal data property of the String prototype object is the standard built-in Object prototype object (15.2.4).

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 data property that has been initialised 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 data property, then
   a. Let *s* be the value of *value's* [[StringData]] internal data property.
   b. If *s* is not **undefined**, then return *s*.
3. Throw a **TypeError** exception.

The phrase "this String value" within the specification of a method refers to the result returned by calling the abstract operation thisStringValue with the **this** value of the method invocation passed as the argument.

15.5.3.1 String.prototype.constructor

The initial value of **`String.prototype.constructor`** is the built-in **`String`** constructor.

Deleted: *elements*
Deleted: *elements*
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: *elements*
Deleted: Quasi
Deleted: quasi
Deleted: false
Deleted: z
Deleted: String
Deleted: (its [[Class]] is **`"String"`**)
Deleted: whose value is an empty String
Deleted: The String prototype object has the [[IsStringNativeBrandBuiltinBrand]] internal d property with value StringWrapperBuiltinStringWrapper.
Deleted: z
Deleted: *n*

15.5.3.2 String.prototype.toString ( )

When the `toString` method is called, the following steps are taken:

1. Let *s* be thisStringValue(**this** value).
2. Return *s*.

NOTE For a String object, the `toString` method happens to return the same thing as the `valueOf` method.

15.5.3.3 String.prototype.valueOf ( )

When the `valueOf` method is called, the following steps are taken:

1. Let *s* be thisStringValue(**this** value).
2. Return *s*.

15.5.3.4 String.prototype.charAt (pos)

NOTE Returns a single element String containing the code unit at element position *pos* in the String value resulting from converting this object to a String. If there is no element at that position, the result is the empty String. The result is a String value, not a String object.

If *pos* is a value of Number type that is an integer, then the result of `x.charAt(`*pos*`)` is equal to the result of `x.substring(`*pos*`, `*pos*`+1)`.

When the `charAt` method is called with one argument *pos*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *position* be ToInteger(*pos*).
5. ReturnIfAbrupt(*position*).
6. Let *size* be the number of elements in *S*.
7. If *position* < 0 or *position* ≥ *size*, return the empty String.
8. Return a String of length 1, containing one code unit from *S*, namely the code unit at position *position*, where the first (leftmost) code unit in *S* is considered to be at position 0, the next one at position 1, and so on.

NOTE The `charAt` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.5 String.prototype.charCodeAt (pos)

NOTE Returns a Number (a nonnegative integer less than 2^{16}) that is the code unit value of the string element at position *pos* in the String resulting from converting this object to a String. If there is no element at that position, the result is **NaN**.

When the `charCodeAt` method is called with one argument *pos*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *position* be ToInteger(*pos*).
5. ReturnIfAbrupt(*position*).
6. Let *size* be the number of elements in *S*.
7. If *position* < 0 or *position* ≥ *size*, return **NaN**.
8. Return a value of Number type, whose value is the code unit value of the element at position *position* in the String *S*, where the first (leftmost) element in *S* is considered to be at position 0, the next one at position 1, and so on.

Deleted: Returns this String value. (

Deleted: ote

Formatted: Note

Deleted: that, f

Deleted:)

Deleted: The `toString` function is not generic; it throws a **TypeError** exception if its **this** value is not a String or a String object. Therefore, it cannot be transferred to other kinds of objects for use as a metho

Comment [AWB9168]: Perhaps BMPCharAt should provided as an alias for this method and charAt should marked as obsolete.

Deleted: Returns this String value.¶
The `valueOf` function is not generic; it throws a **TypeError** exception if its **this** value is not a String or String object. Therefore, it cannot be transferred to oth kinds of objects for use as a method.¶

Formatted: Note

Deleted: character

Deleted: String

Deleted: character

Deleted: ReturnIfAbrupt(

Deleted: Call

Deleted: passing the

Deleted: as its argument

Deleted:)

Deleted: the result of calling

Deleted: , giving it the **this** value as its argument

Deleted: *s*

Deleted: characters

Deleted: character

Deleted: character

Deleted: character

Formatted: Note

Deleted: representing

Deleted: character

Deleted: character

Deleted: ReturnIfAbrupt(

Deleted:)

Deleted: Call CheckObjectCoercible passing the **this** val as its argument.

Deleted: the result of calling

Deleted: , giving it the **this** value as its argument

Deleted: characters

Deleted: character

Deleted: character

NOTE The **charCodeAt** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.3.6 String.prototype.concat (...args)

NOTE When the **concat** method is called with zero or more arguments, it returns a String consisting of the string elements of this object (converted to a String) followed by the string elements of each of the arguments converted to a String. The result is a String value, not a String object.

The following steps are taken:

1. Assert: *args* is a well-formed rest parameter object.
2. Let *O* be CheckObjectCoercible(**this** value).
3. Let *S* be ToString(*O*).
4. ReturnIfAbrupt(*S*).
5. Let *args* be an internal list that is a copy of the argument list passed to this function.
6. Let *R* be *S*.
7. Repeat, while *args* is not empty
   a. Remove the first element from *args* and let *next* be the value of that element.
   b. Let *nextString* be ToString(*next*)
   c. ReturnIfAbrupt(*nextString*).
   d. Let *R* be the String value consisting of the string elements in the previous value of *R* followed by the string elements of *nextString*.
8. Return *R*.

Deleted: [string1 [, string2 [, …]]]
Deleted: …
Deleted: *string1*, *string2*, etc.
Deleted: characters
Formatted: Note
Deleted: characters
Deleted: *string1*, *string2*, etc. (where each argument is
Deleted:)
Deleted: .
Deleted: *strings*
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: characters
Deleted: characters
Deleted: ToString(
Deleted:)

The **length** property of the **concat** method is **1**.

NOTE The **concat** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.3.7 String.prototype.indexOf (searchString, position)

If *searchString* appears as a substring of the result of converting this object to a String, at one or more positions that are greater than or equal to *position*, then the index of the smallest such position is returned; otherwise, **-1** is returned. If *position* is **undefined**, 0 is assumed, so as to search all of the String.

The **indexOf** method takes two arguments, *searchString* and *position*, and performs the following steps:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be ToString(*searchString*).
5. ReturnIfAbrupt(*searchString*).
6. Let *pos* be ToInteger(*position*). (If *position* is **undefined**, this step produces the value **0**).
7. ReturnIfAbrupt(*pos*).
8. Let *len* be the number of elements in *S*.
9. Let *start* be min(max(*pos*, 0), *len*).
10. Let *searchLen* be the number of elements in *searchStr*.
11. Return the smallest possible integer *k* not smaller than *start* such that *k+ searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the code unit at position *k+j* of *S* is the same as the code unit at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: characters
Deleted: characters
Deleted: character
Deleted: character

The **length** property of the **indexOf** method is **1**.

NOTE The **indexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.8 String.prototype.lastIndexOf (searchString, position)

If *searchString* appears as a substring of the result of converting this object to a String at one or more positions that are smaller than or equal to *position*, then the index of the greatest such position is returned; otherwise, **-1** is returned. If *position* is **undefined**, the length of the String value is assumed, so as to search all of the String.

The **lastIndexOf** method takes two arguments, *searchString* and *position*, and performs the following steps:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be ToString(*searchString*).
5. ReturnIfAbrupt(*searchString*).
6. Let *numPos* be ToNumber(*position*). (If *position* is **undefined**, this step produces the value **NaN**).
7. ReturnIfAbrupt(*numPos*).
8. If *numPos* is **NaN**, let *pos* be $+\infty$; otherwise, let *pos* be ToInteger(*numPos*).
9. Let *len* be the number of elements in *S*.
10. Let *start* be min(max(*pos*, 0), *len*).
11. Let *searchLen* be the number of elements in *searchStr*.
12. Return the largest possible nonnegative integer *k* not larger than *start* such that *k*+ *searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the code unit at position *k*+*j* of *S* is the same as the code unit at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing the **this** val as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argument
Deleted: characters
Deleted: characters
Deleted: character
Deleted: character

The **length** property of the **lastIndexOf** method is **1**.

NOTE The **lastIndexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.9 String.prototype.localeCompare (that)

When the **localeCompare** method is called with one 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 the system default locale, and will be negative, zero, or positive, depending on whether *S* comes before *That* in the sort order, the Strings are equal, or *S* comes after *That* in the sort order, respectively.

Before perform the comparisons the following steps are performed to prepare the Strings:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *That* be ToString(*that*).
5. ReturnIfAbrupt(*That*).

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing the **this** val as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argument
Deleted: 15.4.4

The **localeCompare** method, if considered as a function of two arguments **this** and *that*, is a consistent comparison function (as defined in 15.4.3.11) 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 and to return **0** when comparing Strings that are considered canonically equivalent by the Unicode standard.

If no language-sensitive comparison at all is available from the host environment, this function may perform a bitwise comparison.

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. It is strongly recommended that this function treat Strings that are canonically equivalent according to the Unicode standard as identical (in other words, compare the Strings as if they had both been converted to Normalised Form C or D first). It is also recommended that this function not honour Unicode compatibility equivalences or decompositions.

NOTE 3 The second parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 4 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.

15.5.3.10 String.prototype.match (regexp)

When the **`match`** method is called with argument *regexp*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. If Type(*regexp*) is Object and HasProperty(*regexp*, @@isRegExp) is **true**, then let *rx* be *regexp*;
5. Else, let *rx* be the result of the abstract operation RegExpCreate (15.10.4.1) with arguments *regexp* and **undefined**.
6. ReturnIfAbrupt(*rx*).
7. Return the result of Invoke(*rx*, **`"match"`**, (*S*)).

NOTE The **`match`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.11 String.prototype.replace (searchValue, replaceValue)

When the **`replace`** method is called with arguments *searchValue* and *replaceValue* the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *string* be ToString(*O*).
3. ReturnIfAbrupt(*string*).
4. If Type(*searchValue*) is Object and HasProperty(*searchValue*, @@isRegExp) is **true**, then
   a. Return the result of Invoke(*searchValue*, **`"replace"`**, (*string*, *replaceValue*)).
5. Let *searchString* be ToString(*searchValue*).
6. ReturnIfAbrupt(*searchString*).
7. Search *string* for the first occurrence of *searchString* and let *pos* be the index position within *string* of the first code unit of the matched substring and let *matched* be *searchString*. If no occurrences of *searchString* were found, return *string*.
8. If IsCallable(*replaceValue*) is **true**, then
   a. Let *replValue* be the result of calling the [[Call]] internal methods of *replaceValue* passing **undefined** as the **this** value and a List containing *matched*, *pos*, and *string* as the argument list.
   b. Let *replStr* be ToString(*replValue*).
   c. ReturnIfAbrupt(*replStr*).
9. Else,
   a. Let *captures* be an empty List.
   b. Let *replStr* be the result of the abstract operation GetReplaceSubstitution(*matched*, *string*, *pos*, *captures*).
10. Let *tailPos* be *pos* + the number of code units in *matched*.
11. Let *newString* be the String formed by concatenating the first *pos* code units of *string*, *replString*, 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*.

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: the value of the [[Class]] internal property of
Deleted: is **`"RegExp"`**has thea [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp
Deleted: abstraction operation
Deleted: a new RegExp object created as if by the expression **`new RegExp(`***regexp***`)`** where **`RegExp`** is the standard built-in constructor wit that name.
Deleted: <#>Let *args* be a new List with *S* as sole element.¶
Deleted: <#>Let *global* be the result of callin the [[Get]] internal method of(*rx* with argument **`"global"`**).¶ <#>ReturnIfAbrupt(*global*).¶
Deleted:
Deleted: *regexp*
Deleted: *args*
Deleted: *A*
Deleted: First set *string* according to the
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: <#>Let *args* be a new List with
Deleted: *regexp*
Deleted: args
Deleted: <#>If *searchValue* is a regular
Deleted: <#>Assert: If *searchValue* is not a
Deleted: l
Deleted: and
Deleted: search *string* for the first occurrence
Deleted: let *pos* be 0 and let *matched* be the
Deleted: Let *m* be 0.
Deleted:
Deleted: is a function
Deleted: then
Deleted: V
Deleted: for
Deleted: each matched substring, call the

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.

Runtime Semantics: GetReplaceSubstitution Abstract Operation

The abstract operation GetReplaceSubstitution(*matched*, *string*, *position*, *captures*) performs the following steps:

1. Assert: Type(*matched*) is String.
2. Let *matchLength* be the number of code units in *matched*.
3. Assert: Type(*string*) is String.
4. Let *stringLength* be the number of code units in *string*.
5. Assert: *position* is a non-negative integer.
6. Assert: *position* ≤ *stringLength*.
7. Assert: *captures* is a possibly empty List of Strings.
8. Let *tailPos* be *position* + *matchLength*.
9. Let *m* be the number of elements in *captures*.
10. Let *result* be a String value derived from *matched* by replacing code unit elements in *matched* by replacement text as specified in Table 34. These `$` replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements.
11. Return *result*.

Table 34 — Replacement Text Symbol Substitutions

Code unit	Unicode Characters	Replacement text
0x0024, 0x0024	`$$`	`$`
0x0024, 0x0026	`$&`	*matched*
0x0024, 0x0060	``$` ``	If *position* is 0, the replacement is the empty String. Otherwise the replacement is the substring of *string* that starts at index 0 and whose last code point is at index *position* -1.
0x0024, 0x0027	`$'`	If *tailPos* ≥ *stringLength*, the replacement is the empty String. Otherwise the replacement is the substring of *string* that starts at index *tailPos* and continues to the end of *string*.
0x0024, N where 0x0031 ≤ N ≤ 0x0039	`$n` where `n` is one of `1 2 3 4 5 6 7 8 9` and `$n` is not followed by a decimal digit	The *n*th element of *captures*, where *n* is a single digit in the range 1 to 9. If *n*≤*m* and the *n*th element of *captures* is **undefined**, use the empty String instead. If *n*>*m*, the result is implementation-defined.
0x0024, N, N where 0x0030 ≤ N ≤ 0x0039	`$nn` where `n` is one of `0 1 2 3 4 5 6 7 8 9`	The *nn*th elemet of *captures*, where *nn* is a two-digit decimal number in the range 01 to 99. If *nn*≤*m* and the *nn*th element of *captures* is **undefined**, use the empty String instead. If *nn* is 00 or *nn*>*m*, the result is implementation-defined.
0x0024	`$` in any context that does not match on of the above.	`$`

15.5.3.12 String.prototype.search (regexp)

When the search method is called with argument *regexp*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *string* be ToString(*O*).
3. ReturnIfAbrupt(*string*).
4. If Type(*regexp*) is Object and HasProperty(*regexp*, @@isRegExp) is **true**, then,
    a. Let *rx* be *regexp*;
5. Else,
    a. Let *rx* be the result of the abstract operation RegExpCreate (15.10.3.1) with arguments *regexp* and **undefined**.
6. ReturnIfAbrupt(*rx*).
7. Return the result of Invoke(*rx*, `"search"`, (*string*)).

Deleted: Let *result* be a String value derived from *matched* by replacing code unit elements in *matched* by replacement as specified in ¶
<#>Table 33. These `$` replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements.¶

Deleted: If Type(*value*) is Object and *value* has a [[StringData]] internal data property, then¶
¶

Deleted: The

Deleted: substring.

Deleted: $`

Deleted: r

Deleted: The

Deleted: portion of

Deleted: that precedes the matched substring

Deleted: '

Deleted: r

Deleted: .

Deleted: The portion of *string* that follows the matched

Deleted: and $*n* is not followed by a decimal digit

Deleted: 0

Deleted: 0

Comment [AWB14169]: Why is this impl defined? C

Comment [AWB14170]: "12345".replace("1",functio

Deleted: NOTE The `replace` function is intentional

Deleted: ReturnIfAbrupt(

Deleted:)

Deleted: Call CheckObjectCoercible passing the **this** val

Deleted: the result of calling

Deleted: , giving it the **this** value as its argument

Deleted: *S*

Deleted: the value of the [[Class]] internal property of

Deleted: *regexp* has thea

Deleted: is `"RegExp"`

Deleted:

Deleted: 1

Deleted: ,

Deleted: let

Deleted: abstraction operation

Deleted: 4

Deleted: a new RegExp object created as if by the

Deleted: <#>Let *args* be a new List with the single

Deleted: <#>Search the value *string* from its beginning

Deleted: *ege*

Deleted: *p*

Deleted: *args*

Deleted: *result*

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.

15.5.3.13 String.prototype.slice (start, end)

The `slice` method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a String, starting from element position *start* and running to, but not including, element position *end* (or through the end of the String if *end* is **undefined**). If *start* is negative, it is treated as *sourceLength*+*start* where *sourceLength* is the length of the String. If *end* is negative, it is treated as *sourceLength*+*end* where *sourceLength* is the length of the String. The result is a String value, not a String object. The following steps are taken:

Deleted: character

Deleted: character

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *len* be the number of elements in *S*.
5. Let *intStart* be ToInteger(*start*).
6. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ToInteger(*end*).
7. If *intStart* is negative, let *from* be max(*len* + *intStart*,0); else let *from* be min(*intStart*, *len*).
8. If *intEnd* is negative, let *to* be max(*len* + *intEnd*,0); else let *to* be min(*intEnd*, *len*).
9. Let *span* be max(*to* – *from*,0).
10. Return a String value containing *span* consecutive elements from *S* beginning with the element at position *from*.

Deleted: ReturnIfAbrupt(

Deleted:)

Deleted: Call CheckObjectCoercible passing t **this** value as its argument.

Deleted: the result of calling

Deleted: , giving it the **this** value as its argum

Deleted: characters

Deleted: characters

Deleted: character

The `length` property of the `slice` method is **2**.

NOTE The `slice` function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.3.14 String.prototype.split (separator, limit)

Returns an Array object into which substrings of the result of converting this object to a String have been stored. The substrings are determined by searching from left to right for occurrences of *separator*; these occurrences are not part of any substring in the returned array, but serve to divide up the String value. The value of *separator* may be a String of any length or it may be a RegExp object.

Deleted: (i.e., an object with thea [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp whose [[Class]] internal property is `"RegExp"`; see 15.10)

The value of *separator* may be an empty String, an empty regular expression, or a regular expression that can match an empty String. In this case, *separator* does not match the empty substring at the beginning or end of the input String, nor does it match the empty substring at the end of the previous separator match. (For example, if *separator* is the empty String, the String is split up into individual code unit elements; the length of the result array equals the length of the String, and each substring contains one code unit.) If *separator* is a regular expression, only the first match at a given position of the **this** String is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, `"ab".split(/a*?/)` evaluates to the array `["a","b"]`, while `"ab".split(/a*/)` evaluates to the array `["","b"]`.)

Deleted: characters

Deleted: character

If the **this** object is (or converts to) the empty String, the result depends on whether *separator* can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

If *separator* is a regular expression that contains capturing parentheses, then each time *separator* is matched the results (including any **undefined** results) of the capturing parentheses are spliced into the output array. For example,

```
"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/)
```

evaluates to the array

```
["A", undefined, "B", "bold", "/", "B", "and", undefined,
 "CODE", "coded", "/", "CODE", ""]
```

If *separator* is **undefined**, then the result array contains just one String, which is the **this** value (converted to a String). If *limit* is not **undefined**, then the output array is truncated so that it contains no more than *limit* elements.

When the **`split`** method is called, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. ReturnIfAbrupt(*O*).
3. If Type(*separator*) is Object and HasProperty(*separator*, @@isRegExp) is **true** , then,
    a. Return the result of Invoke(*separator*, **`"split"`**, (*O*, *limit*))
4. Let *S* be ToString(*O*).
5. ReturnIfAbrupt(*S*).
6. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
7. Let *lengthA* be 0.
8. If *limit* is **undefined**, let *lim* = 2^{32}–1; else let *lim* = ToUint32(*limit*).
9. Let *s* be the number of elements in *S*.
10. Let *p* = 0.
11. Let *R* be ToString(*separator*).
12. ReturnIfAbrupt(*R*).
13. If *lim* = 0, return *A*.
14. If *separator* is **undefined**, then
    a. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments **`"0"`** and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    b. Assert: *status* is not an abrupt completion.
    c. Return *A*.
15. If *s* = 0, then
    a. Let *z* be the result of SplitMatch(*S*, 0, *R*).
    b. If *z* is not **false**, return *A*.
    c. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments **`"0"`** and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    d. Assert: *status* is not an abrupt completion.
    e. Return *A*.
16. Let *q* = *p*.
17. Repeat, while *q* ≠ *s*
    a. Let *e* be the result of *SplitMatch*(*S*, *q*, *R*).
    b. If *e* is **false**, then let *q* = *q*+1.
    c. Else, *e* is an integer index into *S*,
        i. If *e* = *p*, then let *q* = *q*+1.
        ii. Else, *e* ≠ *p*,
            1. Let *T* be a String value equal to the substring of *S* consisting of the code units at positions *p* (inclusive) through *q* (exclusive).
            2. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            3. Assert: *status* is not an abrupt completion.
            4. Increment *lengthA* by 1.
            5. If *lengthA* = *lim*, return *A*.
            6. Let *p* = *e*.
            7. Let *q* = *p*.
18. Let *T* be a String value equal to the substring of *S* consisting of the code units at positions *p* (inclusive) through *s* (exclusive).
19. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
20. *status* is not an abrupt completion.
21. Return *A*.

Runtime Semantics: SplitMatch Abstract Operation

Deleted: ReturnIfAbrupt(…heckObjectCoercible(**this**

Deleted: Call CheckObjectCoercible passing the **this** val as its argument.

Deleted: <#>Let *S* be the result of calling ToString, givi it the **this** value as its argument.¶

Deleted: *e*

Deleted: <#>Let *args* be a new List with elements *S*(and *limit*.¶

Deleted: *e*

Deleted: *args*

Deleted: a new array created as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructo with that name

Deleted: characters

Deleted: If *separator* has thea [[IsRegExpNativeBrandBuiltinBrand]] internal data proper whose value is **NativeRegExpBuiltinRegExp** is a RegExp object (its [[Class]] is **`"RegExp"`**),let *R* be *separator* let *R* be *separator*; otherwise l

Deleted: =

Deleted: *sepearator*

Deleted: Call

Deleted: **`"`**, … and Property Descriptor {[[Value]]: *S*,

Deleted: the previous step will never result in an abru completion

Deleted: Call …plitMatch(*S*, 0, *R*) and let *z* be its

Deleted: **failure**

Deleted: Call the

Deleted: **`"`**, … and Property Descriptor {[[Value]]: *S*,

Deleted: the previous step will never result in an abru completion

Deleted: Call …et *e* be the result of *SplitMatch*(*S*, *q*, *R*

Deleted: *z* … is **failure**

Deleted: ,

Deleted: *z* … is not **failure**

Deleted: <#>*z* must be a State. Let *e* be *z*'s *endInd* and let *cap* be *z*'s *captures* array.¶

Deleted: ,

Deleted: characters

Deleted: elements

Deleted: Call the

Deleted:), … and Property Descriptor {[[Value

Deleted: the previous step will never result in an abrupt completion

Deleted: <#>Let *i* = 0.¶

Deleted: characters

Deleted: elements

Deleted: Call the

Deleted:), … and Property Descriptor {[[Value]]: *T*,

Deleted: Assert: the previous step will never result in an

The abstract operation *SplitMatch* takes three parameters, a String *S*, an integer *q*, and a String *R*, and performs the following in order to return either **false** or the end index of a match:

1. Type(*R*) must be String. Let *r* be the number of code units in *R*.
2. Let *s* be the number of code units in *S*.
3. If *q*+*r* > *s* then return **false**.
4. If there exists an integer *i* between 0 (inclusive) and *r* (exclusive) such that the code unit at position *q*+*i* of *S* is different from the code unit at position *i* of *R*, then return **false**.
5. Return *q*+*r*.

The **`length`** property of the **`split`** method is **2**.

NOTE The **`split`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.15 String.prototype.substring (start, end)

The **substring** method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a String, starting from element position *start* and running to, but not including, element position *end* of the String (or through the end of the String is *end* is **undefined**). The result is a String value, not a String object.

If either argument is **NaN** or negative, it is replaced with zero; if either argument is larger than the length of the String, it is replaced with the length of the String.

If *start* is larger than *end*, they are swapped.

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *len* be the number of elements in *S*.
5. Let *intStart* be ToInteger(*start*).
6. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ToInteger(*end*).
7. Let *finalStart* be min(max(*intStart*, 0), *len*).
8. Let *finalEnd* be min(max(*intEnd*, 0), *len*).
9. Let *from* be min(*finalStart*, *finalEnd*).
10. Let *to* be max(*finalStart*, *finalEnd*).
11. Return a String whose length is *to* - *from*, containing code units from *S*, namely the code units with indices *from* through *to* −1, in ascending order.

The **`length`** property of the **`substring`** method is **2**.

NOTE The **`substring`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.16 String.prototype.toLowerCase ( )

This function interprets a string value as a sequence of code points, as described in 8.4. The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *cpList* be a List containing in order the code points as defned in 8.4 of *S*, starting at the first element of *S*.
5. For each code point *c* in *cpList*, if the Unicode Character Database provides a language insensitive lower case equivalent of *c* then replace *c* in *cpList* with that equivalent code point(s).

Deleted: or RegExp
Deleted: a MatchResult (see 15.10.2.1)
Deleted: <#>If *R* has thea [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp is a RegExp objec (its [[Class]] is **"RegExp"**), then¶
<#>Call the [[Match]] internal method of *R* giving it the arguments *S* and *q*, and return th MatchResult result.¶
Deleted: characters
Deleted: elements
Deleted: characters
Deleted: elements
Deleted: **the MatchResult failure**
Deleted: character
Deleted: character
Deleted: **failure**
Deleted: <#>Let *cap* be an empty array of captures (see 15.10.2.1).¶
Deleted: the State (
Deleted: , *cap*)
Deleted: (see 15.10.2.1)
Deleted: NOTE 1 The **`split`** method ignor the value of separator **`.global`** for separator that are RegExp objects.¶
Deleted: 2
Deleted: character
Deleted: character
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: characters
Deleted: characters
Deleted: characters
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum

6. Let *cuList* be a new List.
7. For each code point *c* in *cpList*, in order, append to *cuList* the elements of the UTF-16 Encoding (clause 6) of *c*.
8. Let *L* be a String whose elements are, in order, the elements of *cuList* .
9. Return *L*.

Deleted: replace

Deleted: where each character of *L* is either the Unicode lowercase equivalent of the corresponding character of *S* or the actual corresponding character of *S* if no Unicode lowercase equivalent exists

The result must be derived according to the case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file, but also the SpecialCasings.txt file that accompanies it).

Deleted: For the purposes of this operation, the 16-bit code units of the Strings are treated as code points in the Unicode Basic Multilingual Plane. Surrogate code points are directly transferred from *S* to *L* without any mapping

Deleted: in Unicode 2.1.8 and later

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()`.

Deleted: characters

Deleted: characters

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.

15.5.3.17 String.prototype.toLocaleLowerCase ( )

This function interprets a string value as a sequence of code points, as described in 8.4.

This function works exactly the same as `toLowerCase` except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleLowerCase` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.18 String.prototype.toUpperCase ( )

This function interprets a string value as a sequence of code points, as described in 8.4.

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.

Deleted: characters

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.

15.5.3.19 String.prototype.toLocaleUpperCase ( )

This function interprets a string value as a sequence of code points, as described in 8.4.

This function works exactly the same as `toUpperCase` except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleUpperCase` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.20 String.prototype.trim ( )

This function interprets a string value as a sequence of code points, as described in 8.4.

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *T* be a String value that is a copy of *S* with both leading and trailing white space removed. The definition of white space is the union of *WhiteSpace* and *LineTerminator*. When determining whether a Unicode character is in Unicode general category "Zs", code unit sequences are interpreted as UTF-16 encoded code point sequences as specified in 8.4.
5. Return *T*.

NOTE The `trim` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.21 String.prototype.repeat (count)

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *n* be the result of calling ToInteger(*count*).
5. ReturnIfAbrupt(*n*).
6. If *n* < 0, then throw a **RangeError** exception.
7. If *n* is +∞, then throw a **RangeError** exception.
8. Let *T* be a String value that is made from *n* copies of *S* appended together. If *n* is 0, *T* is the empty String.
9. Return *T*.

NOTE 1 This method creates a String consisting of the string elements of this object (converted to String) repeated *count* time.

NOTE 2 The `repeat` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.3.22 String.prototype.startsWith (searchString [, position])

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be ToString(*searchString*).
5. ReturnIfAbrupt(*searchStr*).
6. Let *pos* be ToInteger(*position*). (If *position* is **undefined**, this step produces the value **0**).
7. ReturnIfAbrupt(*pos*).
8. Let *len* be the number of elements in *S*.
9. Let *start* be min(max(*pos*, 0), *len*).
10. Let *searchLength* be the number of elements in *searchString*.
11. If *searchLength*+*start* is greater than *len*, return **false**.
12. If the *searchLength* sequence of elements of *S* starting at *start* is the same as the full element sequence of *searchString*, return **true**.
13. Otherwise, return **false**.

The **length** property of the **startsWith** method is **1**.

NOTE 1 This method returns **true** if the sequence of elements of *searchString* converted to a String is the same as the corresponding elements of this object (converted to a String) starting at *position*. Otherwise returns **false**.

NOTE 2 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.

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call CheckObjectCoercible passing t **this** value as its argument.
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: ≤
Deleted: rh
Deleted: E
Deleted: characters
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Deleted: characters
Deleted: characters
Deleted: characters
Deleted: character
Deleted: characters
Deleted: characters

15.5.3.23 String.prototype.endsWith (searchString [, endPosition])

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be ToString(*searchString*).
5. ReturnIfAbrupt(*searchStr*).
6. Let *len* be the number of elements in *S*.
7. If *endPosition* is **undefined**, let *pos* be *len*, else let *pos* be ToInteger(*endPosition*).
8. ReturnIfAbrupt(*pos*).
9. Let *end* be min(max(*pos*, 0), *len*).
10. Let *searchLength* be the number of elements in *searchString*.
11. Let *start* be *end* - *searchLength*.
12. If *start* is less than 0, return **false**.
13. If the *searchLength* sequence of elements of *S* starting at *start* is the same as the full element sequence of *searchString*, return **true**.
14. Otherwise, return **false**.

The **length** property of the **endsWith** method is **1**.

NOTE 1 Returns **true** if the sequence of elements of *searchString* converted to a String is the same as the corresponding elements of this object (converted to a String) starting at *endPosition* – length(this). Otherwise returns **false**.

NOTE 2 The **endsWith** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of calling
Deleted: , giving it the **this** value as its argument
Deleted: characters
Deleted: characters
Deleted: characters
Deleted: character
Deleted: characters
Deleted: characters

15.5.3.24 String.prototype.contains (searchString, position = 0)

The **contains** method takes two arguments, *searchString* and *position*, and performs the following steps:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be ToString(*searchString*).
5. ReturnIfAbrupt(*searchStr*).
6. Let *pos* be ToInteger(*position*). (If *position* is **undefined**, this step produces the value **0**).
7. ReturnIfAbrupt(*pos*).
8. Let *len* be the number of elements in *S*.
9. Let *start* be min(max(*pos*, 0), *len*).
10. Let *searchLen* be the number of characters in *searchStr*.
11. If there exists any integer *k* not smaller than *start* such that *k* + *searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the character at position *k+j* of *S* is the same as the character at position *j* of *searchStr*, return **true**; but if there is no such integer *k*, return **false**.

The **length** property of the **contains** method is **1**.

NOTE 1 If *searchString* appears as a substring of the result of converting this object to a String, at one or more positions that are greater than or equal to *position*, then return **true**; otherwise, returns **false**. If *position* is **undefined**, 0 is assumed, so as to search all of the String.

NOTE 2 The **contains** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

Deleted: [
Deleted:]
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of calling
Deleted: , giving it the **this** value as its argument
Deleted: characters

15.5.3.25 String.prototype.codePointAt (pos)

NOTE Returns a Number (a nonnegative integer less than 1114112) that is the UTF-16 encoded code point value starting at the string element at position *pos* in the String resulting from converting this object to a String. If there is no

Comment [AWB9171]: Do we also need UnicodeCharAt that returns a string of length 1 or 2?

element at that position, the result is **NaN**. If a valid UTF-16 surrogate pair does not begin at *pos*, the result is the code unit at *pos*.

When the **`codePointAt`** method is called with one argument *pos*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *position* be ToInteger(*pos*).
5. ReturnIfAbrupt(*position*).
6. Let *size* be the number of elements in *S*.
7. If *position* < 0 or *position* ≥ *size*, return **undefined**.
8. Let *first* be the code unit value of the element at index *position* in the String *S*.
9. If *first* < 0xD800 or *first* > 0xDBFF or *position*+1 = *size*, then return *first*.
10. Let *second* be the code unit value of the element at index *position*+1 in the String *S*.
11. If *second* < 0xDC00 or *second* > 0xDFFF, then return *first*.
12. Return ((*first* – 0xD800) × 1024) + (*second* – 0xDC00) + 0x10000.

NOTE The **`codePointAt`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.3.26 String.prototype.normalize (form = `"NFC"`)

When the **`normalize`** method is called with one argument *form*, the following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. If *form* is not provided or **undefined** let *form* be `"NFC"`.
5. Let *f* be ToString(*form*).
6. ReturnIfAbrupt(*f*).
7. If *f* is not one of `"NFC"`, `"NFD"`, `"NFKC"`, or `"NFKD"`, then throw a **RangeError** Exception.
8. Let *ns* be the String value is the result of normalizing *S* into the normalization form named by *f* as specified in UTR 15, Unicode Normalization Forms.
9. 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.

15.5.4 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 data property.

String instances have a **`length`** property, and a set of enumerable properties with array index names.

15.5.4.1 length

The number of elements in the String value represented by this String object.

Once a String object is initialised, this property is unchanging. It has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

Deleted: dsa
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of calling
Deleted: , giving it the **this** value as its argum
Comment [AWB9172]: Note that this is inconsistent with the pre-exisitng behaviour o charCodeAt which returns NaN for out of bounds positions
Deleted: NaN
Deleted: position
Deleted: ., where the first (leftmost) element i is considered to be at position 0, the next one at position 1, and so on.
Deleted: position
Deleted: *first*
Deleted: `charCode`
Deleted:)
Deleted: C
Deleted: then let *form* be `"NFKD"`,
Deleted: by
Deleted: <#>15.5.4.25 String.prototyp oArray()¶
<#>The following steps are taken:¶
<#>ReturnIfAbrupt(CheckObjectCoerci e(this value)).¶
<#>Let *S* be the result of calling ToStri giving it the this value as its argument.¶
<#>ReturnIfAbrupt(*S*).¶
<#>Let *len* be the number of characters in *S*.¶
<#>Let *array* be the result of the abstraction operation ArrayCreate (15.4 with argument *len*.¶
<#>Let *n* be 0¶
<#>Repeat, while *n* < *len*:¶
<#>Let *c* be the character at position *n* *S*.¶
<#>Call the [[DefineOwnProperty]] internal method of *array* with argument ToString(*n*), the PropertyDescriptor {[[Value]]: *c*, [[Writable]]: true,
Deleted: and internal data properties
Deleted:
Deleted: and their [[Class]]have a [[IsStringNativeBrand]] internal property with
Deleted: value is `"String"`
Deleted: .
Deleted: also
Deleted: a [[PrimitiveValue]] internal propert
Deleted: The [[PrimitiveValue]] internal property is the String value represented by th
Deleted: characters
Deleted: created
Deleted: z

15.6 Boolean Objects

15.6.1 The Boolean Constructor

The Boolean constructor is the %Boolean% instrinsic object and the initial value of the **Boolean** property of the global object. When `Boolean` is called as a function rather than as a constructor, it performs a type conversion. However, if the **this** value passed in the call is an Object with an uninitialised [[BooleanData]] internal data property, it initialises the **this** value using the argument value. This permits `Boolean` to be used both to perform type conversion and to perform constructor instance initialization.

The `Boolean` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified `Boolean` behaviour must include a `super` call to the `Boolean` constructor to initialise the [[BooleanData]] state of subclass instances.

15.6.1.1 Boolean (value)

When `Boolean` is called with argument *value*, the following steps are taken:

1. Let *O* be the **this** value.
2. Let *b* be ToBoolean(*value*).
3. If Type(*O*) is Object and *O* has a [[BooleanData]] internal data property and the value of [[BooleanData]] is **undefined**, then
   a. Set the value of *O's* [[BooleanData]] internal data property to *b*.
   b. Return *O*.
4. Return *b*.

15.6.1.2 new Boolean (... argumentsList)

`Boolean` called as part of a new expression , it initialises a newly created object:

1. Let *F* be the `Boolean` function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If `Boolean` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.6.2 Properties of the Boolean Constructor

The value of the [[Prototype]] internal data property of the Boolean constructor is the Function prototype object (15.3.3).

Besides the `length` property (whose value is **1**), the Boolean constructor has the following property:

15.6.2.1 Boolean.prototype

The initial value of `Boolean.prototype` is the Boolean prototype object (15.6.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.6.2.2 Boolean[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%BooleanPrototype%"`, ([[BooleanData]])).
3. Return *obj*.

Deleted: <#>15.5.5.2 [[GetOwnProperty]] (P)
String objects use a variation of the [[GetOwnProperty]] internal method used for other native ECMAScript obje (8.12.1). This special internal method provides access named properties corresponding to the individual characters elements of String objects.
Assume *S* is a String object and *P* is a String.
When the [[GetOwnProperty]] internal method of *S* is called with property name *P*, the following steps are taken:
<#>Let *desc* be the result of calling the default [[GetOwnProperty]] internal method (8.12.1) on *S* with argument *P*.
<#>If *desc* is not **undefined** return *desc*.
<#>If ToString(abs(ToInteger(*P*))) is not the same value a *P*, return **undefined**.
<#>Let *str* be the String value of the [[PrimitiveValue]] internal property of *S*.
<#>Let *index* be ToInteger(*P*).
<#>Let *len* be the number of characters elements in *str*.
<#>If *len* ≤ *index*, return **undefined**.
<#>Let *resultStr* be a String of length 1, containing one character code unit from *str*, specifically the character code unit at position *index*, where the first (leftmost) character element in *str* is considered to be at position 0, the next one position 1, and so on.
<#>Return a Property Descriptor { [[Value]]: *resultStr*, [[Enumerable]]: **true**, [[Writable]]: **false**, [[Configurable]]: **false** }

Deleted: Called as a Function

Deleted: z

Deleted: ize

Deleted: `Boolean` may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

Deleted: Returns a Boolean value (not a Boolean obje computed by ToBoolean(*value*).

Deleted: <#>The Boolean Constructor
When `Boolean` is called as part of a `new` expression it a constructor: it initialises the newly created ordinary object.

Deleted: value

Deleted: with optional argument *value* performs the following steps

Deleted: be the be the

Deleted: a

Deleted: The [[Prototype]] internal data property of the newly constructed object is set to the original Boolean prototype object, the one that is the initial value of `Boolean.prototype` (15.6.3.1).
The [[Class]] internal property of the newly constructed Boolean object is set to `"Boolean"` has a [[IsBooleanNativeBrandBuiltinBrand]] internal data property with value BooleanWrapperBuiltinBooleanWrapper.
The [[PrimitiveValueBooleanValueData]] internal data property of the newly constructed Boolean object is set ToBoolean(*value*).
The [[Extensible]] internal data property of the newly constructed object is set to **true**.

Deleted: 15.3.3

Deleted: the internal properties and

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[BooleanData]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the Boolean constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.6.3 Properties of the Boolean Prototype Object

The Boolean prototype object is an ordinary object. It is not a Boolean instance and does not have a [[BooleanData]] internal data property.

The value of the [[Prototype]] internal data property of the Boolean prototype object is the standard built-in Object prototype object (15.2.4).

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 data property, then
   a. Let *b* be the value of *value's* [[BooleanData]] internal data property.
   b. If *b* is not **undefined**, then return *b*.
3. Throw a **TypeError** exception.

15.6.3.1 Boolean.prototype.constructor

The initial value of `Boolean.prototype.constructor` is the built-in `Boolean` constructor.

15.6.3.2 Boolean.prototype.toString ( )

The following steps are taken:

1. Let *b* be thisBooleanValue(**this** value).
2. ReturnIfAbrupt(*b*).
3. If *b* is **true**, then return `"true"`; else return `"false"`.

15.6.3.3 Boolean.prototype.valueOf ( )

The following steps are taken:

1. Return thisBooleanValue(**this** value).

15.6.4 Properties of Boolean Instances

Boolean instances are ordinary objects that inherit properties from the Boolean prototype object. Boolean instances have a [[BooleanData]] internal data property. The [[BooleanData]] internal data property is the Boolean value represented by this Boolean object.

15.7 Number Objects

15.7.1 The Number Constructor

The Number constructor is the %Number% instrinsic object and the initial value of the `Number` property of the global object. When `Number` is called as a function rather than as a constructor, it performs a type conversion. However, if the **this** value value passed in the call is an Object with an uninitialised [[NumberData]] internal data property, it initialises the **this** value using the argument value. This permits `Number` to be used both to perform type conversion and to perform constructor instance initialization.

The `Number` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified `Number` behaviour must include a `super` call to the `Number` constructor to initialise the [[NumberData]] state of subclass instances.

Deleted: false
Deleted: z
Deleted: itself
Deleted: Boolean
Deleted:
Deleted: (its [[Class]] is `"Boolean"`)
Deleted: whose
Deleted: value
Deleted: Value
Deleted: has the value is **false**
Deleted: The Boolean prototype object has [[IsBooleanNativeBrandBuiltinBrand]] internal data property whose value is BooleanWrapperBuiltinBooleanWrapper.
Deleted: *n*
Deleted: *B*
Deleted: the **this** value
Deleted: <#>If Type(*B*) is Boolean, then let *b* be *B*.¶
<#>Else if Type(*B*) is Object and the value of the [[Class]] internal property of *B* his `"Boolean"` a [[IsBooleanNativeWrapperBooleanData]] internal data property whose value is BooleanWrapper, then let *b* be the value of the [[BooleanDataPrimitiveValue]] internal data property of *B*.¶
<#>Else throw a **TypeError** exception.¶
Deleted: <#>Let *B* be the **this** value.¶
<#>If Type(*B*) is Boolean, then let *b* be *B*.¶
<#>Else if Type(*B*) is Object and the value of the [[Class]] internal property of *B* is `"Boolean"` has a [[IsBooleanNativeWrapperBooleanData]] internal data property whose value is BooleanWrapper, then let *b* be the value of the [[BooleanDataPrimitiveValue]] internal data property of *B*.¶
<#>Else throw a **TypeError** exception.¶
Deleted: *b*
Deleted:
Deleted: and
Deleted: Boolean instance objects their have [[ClassIsBooleanNativeBrandBuiltinBrand]]
Deleted: value is `"Boolean"`
Deleted: .
Deleted: also
Deleted: PrimitiveValue
Deleted: ¶
Deleted: PrimitiveValue
Deleted: Called as a Function
Deleted: z
Deleted: ize
Deleted: `Number` may be subclassed and
Deleted: ize

15.7.1.1 Number ([value])

When **`Number`** is called with argument *number*, the following steps are taken:

1. Let *O* be the **this** value.
2. If no arguments were passed to this function invocation, then let *n* be +0.
3. Else, let *n* be ToNumber(*value*).
4. ReturnIfAbrupt(*n*).
5. If Type(*O*) is Object and *O* has a [[NumberData]] internal data property and the value of [[NumberData]] is **undefined**, then
   a. Set the value of *O's* [[NumberData]] internal data property to *n*.
   b. Return *O*.
6. Return *n*.

15.7.1.2 new Number (...argumentsList)

`Number` called as part of a new expression with argument list *argumentsList* it performs the following steps:

1. Let *F* be the **`Number`** function object on which the **`new`** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **`new`** operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If **`Number`** is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.7.2 Properties of the Number Constructor

The value of the [[Prototype]] internal data property of the Number constructor is the Function prototype object (15.3.3).

Besides the **`length`** property (whose value is **1**), the Number constructor has the following properties:

15.7.2.1 Number.prototype

The initial value of **`Number.prototype`** is the Number prototype object (15.7.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.2.2 Number.MAX_VALUE

The value of **`Number.MAX_VALUE`** is the largest positive finite value of the Number type, which is approximately $1.7976931348623157 \times 10^{308}$.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.2.3 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-764 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** }.

Deleted:

Deleted:

Deleted: Returns a Number value (not a Number obje computed by ToNumber(*value*) if *value* was supplied, els returns **+0**.

Deleted: *value* is **undefined**

Deleted: <#>15.7.2 The Number Constructor¶ When **`Number`** is called as part of a **`new`** expression it i constructor: it initialises the newly created ordinary obj **15.7.2.1**

Deleted: [value

Deleted: args

Deleted:]

Deleted: The [[Prototype]] internal data property of the newly constructed object is set to the original Number prototype object, the one that is the initial value of **`Number.prototype`** (15.7.3.1).¶ The [[Class]] internal property of the newly constructed object is set to **`"Number"`** has a [[IsNumberNativeBrandBuiltinBrand]] internal data property whose value is NumberWrapperBuiltinNumberWrapper.¶ The [[PrimitiveValueNumberData]] internal data proper of the newly constructed object is set to ToNumber(*valu* if *value* was supplied, else to +0.¶ The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶

Deleted: *args*

Deleted: be the be the

Deleted: a

Deleted: 15.3.3

Deleted: the internal properties and

Deleted: precission

Deleted: ed

15.7.2.4 Number.NaN

The value of **`Number.NaN`** is **NaN**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.2.5 Number.NEGATIVE_INFINITY

The value of Number.NEGATIVE_INFINITY is −∞.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.2.6 Number.POSITIVE_INFINITY

The value of Number.POSITIVE_INFINITY is +∞.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.7 Number.EPSILON

The value of Number.EPSILON is the difference between 1 and the smallest value greater than 1 that is representable as a Number value, which is approximately 2.2204460492503130808472633361816 x 10^{-16}.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.8 Number.MAX_INTEGER

The value of Number.MAX_INTEGER is the largest integer value that can be represented as a Number value without losing precision, which is 9007199254740991.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.9 Number.parseInt (string, radix)

Same as 15.1.2.2.

15.7.3.10 Number.parseFloat (string)

Same as 15.1.2.3.

15.7.3.11 Number.isNaN (number)

When the **`Number.isNaN`** is called with one argument *number*, the following steps are taken:

1. If Type(*number*) is not Number, return **false**.
2. If *number* is **NaN**, return **true**.
3. Otherwise, return **false**.

NOTE This function differs from the global isNaN function (15.1.2.4) is that it does not convert its argument to a Number before determining whether it is NaN.

15.7.3.12 Number.isFinite (number)

When the **`Number.isFinite`** is called with one argument *number*, the following steps are taken:

1. If Type(*number*) is not Number, return **false**.
2. If *number* is **NaN**, +∞, or −∞, return **false**.
3. Otherwise, return **true**.

Deleted: 7

Deleted: 8

Comment [AWB7173]: Same specification same value?

Deleted: 9

Comment [AWB7174]: Same specification same value?

Deleted: 0

Deleted: 1

15.7.3.13 Number.isInteger (number)

When the `Number.isInteger` is called with one argument *number*, the following steps are taken:

1. If Type(*number*) is not Number, return **false**.
2. Let *integer* be ToInteger(*number*).
3. If *integer* is not equal to *number*, return **false**.
4. Otherwise, return **true**.

15.7.3.14 Number.toInteger (number)

When the `Number.toInteger` is called with one argument *number*, the following steps are taken:

1. Return ToInteger(*number*).

15.7.2.7 Number[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%NumberPrototype%"`, ([[NumberData]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[NumberData]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the Number constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.7.3 Properties of the Number Prototype Object

The Number prototype object is an ordinary object. It is not a Number instance and does not have a [[NumberData]] internal data property.

The value of the [[Prototype]] internal data property of the Number prototype object is the standard built-in Object prototype object (15.2.4).

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 data property that has been initialised to a Number value.

The abstract operation thisNumberValue(*value*) performs the following steps:

1. If Type(*value*) is Number, return *value*.
2. If Type(*value*) is Object and *value* has a [[NumberData]] internal data property, then
   a. Let *n* be the value of *value's* [[NumberData]] internal data property.
   b. If *n* is not **undefined**, then return *n*.
3. Throw a **TypeError** exception.

The phrase "this Number value" within the specification of a method refers to the result returned by calling the abstract operation thisNumberValue with the **this** value of the method invocation passed as the argument.

15.7.3.1 Number.prototype.constructor

The initial value of `Number.prototype.constructor` is the built-in `Number` constructor.

Deleted: 2

Comment [AWB7175]: Name changed from toInteg… to toInt at Marcdh 29 TC39 meeting. And changed bac… to toInteger at the Jan 29, 2013 meeting.

Deleted: 3

Deleted: `eger`

Deleted: ¶

Deleted: false

Deleted: z

Deleted: itself a Number

Deleted: (its [[Class]] is `"Number"`)

Deleted: with

Deleted: IsNumber

Deleted: NativeBrand

Deleted: BuiltinBrand

Deleted: whose value is NumberWrapperBuiltinNumberWrapper. whose Its valu… is +0

Deleted: Object

Deleted: for which the value of

Deleted: the

Deleted: Class

Deleted: IsNumber

Deleted: NativeBrand

Deleted: z

Deleted: whose value is NumberWrapper

Deleted: is `"Number"`

Deleted: In the following descriptions of functions that are properties of the Number prototype object, the phra… "this Number object" refers to either the object that is th… **this** value for the invocation of the function or, if Type(**this** value) is Number, an object that is created a… if by the expression `new Number(`**this** value`)` where `Number` is the standard built-in constructor with that name. Also, t

Deleted: refers to either the

Deleted: Number value represented by this Number object, that is, the value of the [[PrimitiveValueNumberData]] internal data property of this Number object or the **this** value if its type is Numb… A **TypeError** exception is thrown if the **this** value is neither an object for which the value of thethat has an [[ClassIsNumberNativeBrandNumberData]] internal da… property is `"Number"` or a value whose type is Numbe…

15.7.3.2 Number.prototype.toString ([radix])

The optional *radix* should be an integer value in the inclusive range 2 to 36. If *radix* not present or is **undefined** the Number 10 is used as the value of *radix*. If ToInteger(*radix*) is the Number 10 then this Number value is given as an argument to the ToString abstract operation; the resulting String value is returned.

If ToInteger(*radix*) is not an integer between 2 and 36 inclusive throw a **RangeError** exception. If ToInteger(*radix*) is an integer from 2 to 36, but not 10, the result is a String representation of this Number value using the specified radix. Letters `a-z` are used for digits with values 10 through 35. The precise algorithm is implementation-dependent if the radix is not 10, however the algorithm should be a generalisation of that specified in 9.1.8.1.

The `toString` function is not generic; it throws a **TypeError** exception if its **this** value is not a Number or a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.3.3 Number.prototype.toLocaleString()

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`.

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.7.3.4 Number.prototype.valueOf ( )

1. Let *x* be thisNumberValue(**this** value).
2. Return *x*.

15.7.3.5 Number.prototype.toFixed (fractionDigits)

Note **toFixed** returns a String containing this Number value represented in decimal fixed-point notation with *fractionDigits* digits after the decimal point. If *fractionDigits* is **undefined**, 0 is assumed.

The following steps are performed:

1. Let *x* be thisNumberValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. Let *f* be ToInteger(*fractionDigits*). (If *fractionDigits* is **undefined**, this step produces the value `0`).
4. ReturnIfAbrupt(*f*).
5. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
6. If *x* is **NaN**, return the String `"NaN"`.
7. Let *s* be the empty String.
8. If $x < 0$, then
   a. Let s be `"-"`.
   b. Let x = –x.
9. If $x \geq 10^{21}$, then
   a. Let *m* = ToString(*x*).
10. Else $x < 10^{21}$,
    a. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ 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.
          2. Let *m* be the concatenation of Strings *z* and *m*.
          3. Let $k = f + 1$.

Comment [AWB7176]: TODO: need to provide algorithm that orders abnormal completion detection

Deleted:

Deleted: v

Deleted: Returns this Number value.¶

Deleted: The `valueOf` function is not gener it throws a **TypeError** exception if its **this** val is not a Number or a Number object. Therefo it cannot be transferred to other kinds of obje for use as a method.¶

Deleted: Return

Deleted: .

Deleted: Specifically, perform the following steps

Deleted: <#>Let *x* be this Number value.¶ <#>ReturnIfAbrupt(*x*).¶

Deleted: ,

Deleted: characters

Deleted: character

Deleted: '0'

iii. Let *a* be the first $k–f$ elements of *m*, and let *b* be the remaining *f* elements of *m*.
iv. Let *m* be the concatenation of the three Strings *a*, `"."`, and *b*.

11. Return the concatenation of the Strings *s* and *m*.

The **`length`** property of the **`toFixed`** method is **1**.

If the **`toFixed`** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **`toFixed`** for values of *fractionDigits* less than 0 or greater than 20. In this case **`toFixed`** would not necessarily throw **RangeError** for such values.

NOTE The output of **`toFixed`** may be more precise than **`toString`** for some values because toString only prints enough significant digits to distinguish the number from adjacent number values. For example, **`(1000000000000000128).toString()`** returns **`"1000000000000000100"`**, while **`(1000000000000000128).toFixed(0)`** returns **`"1000000000000000128"`**.

15.7.3.6 Number.prototype.toExponential (fractionDigits)

Return a String containing this Number value represented in decimal exponential notation with one digit before the significand's decimal point and *fractionDigits* digits after the significand's decimal point. If *fractionDigits* is **undefined**, include as many significand digits as necessary to uniquely specify the Number (just like in ToString except that in this case the Number is always output in exponential notation). Specifically, perform the following steps:

1. Let *x* be thisNumberValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. Let *f* be ToInteger(*fractionDigits*).
4. Assert: *f* is 0, when *fractionDigits* is **undefined**.
5. ReturnIfAbrupt(*f*).
6. If *x* is **NaN**, return the String **`"NaN"`**.
7. Let *s* be the empty String.
8. If $x < 0$, then
    a. Let *s* be **`"-"`**.
    b. Let $x = –x$.
9. If $x = +\infty$, then
    a. Return the concatenation of the Strings *s* and **`"Infinity"`**.
10. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
11. If $x = 0$, then
    a. Let *m* be the String consisting of f+1 occurrences of the code unit 0x0030.
    b. Let $e = 0$.
12. Else $x \neq 0$,
    a. If *fractionDigits* is not **undefined**, then
        i. Let *e* and *n* be integers such that $10^f \le n < 10^{f+1}$ and for which the exact mathematical value of $n \times 10^{e–f} – x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e–f}$ is larger.
    b. Else *fractionDigits* is **undefined**,
        i. Let *e*, *n*, and *f* be integers such that $f \ge 0$, $10^f \le n < 10^{f+1}$, the number value for $n \times 10^{e–f}$ is *x*, and *f* is as small as possible. Note that the decimal representation of *n* has f+1 digits, *n* is not divisible by 10, and the least significant digit of *n* is not necessarily uniquely determined by these criteria.
    c. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
13. 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*.
14. If $e = 0$, then
    a. Let c = **`"+"`**.
    b. Let d = **`"0"`**.

Deleted: characters
Deleted: characters
Deleted:
Deleted: value
Deleted: *fractionDigits* is not **undefined** and (
Deleted:)
Deleted: <#>If *fractionalDigits* is **undefined**, then L let *f* = 0.¶
Deleted: character
Deleted: '0'
Deleted: ,
Deleted: ,
Deleted: character
Deleted: characters

15. Else
    a. If $e > 0$, then let *c* = **"+"**.
    b. Else *e* ≤ 0,
        i. Let *c* = **"-"**.
        ii. Let *e* = –*e*.
    c. Let *d* be the String consisting of the digits of the decimal representation of *e* (in order, with no leading zeroes).
16. Let *m* be the concatenation of the four Strings *m*, **"e"**, *c*, and *d*.
17. Return the concatenation of the Strings *s* and *m*.

Deleted: ,

The **length** property of the **toExponential** method is **1**.

If the **toExponential** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **toExponential** for values of *fractionDigits* less than 0 or greater than 20. In this case **toExponential** would not necessarily throw **RangeError** for such values.

NOTE For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 9.b.i be used as a guideline:

i. Let *e*, *n*, and *f* be integers such that $f \ge 0$, $10^f \le n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is x, and *f* is as small as possible. If there are multiple possibilities for *n*, choose the value of *n* for which $n \times 10^{e-f}$ is closest in value to *x*. If there are two such possible values of *n*, choose the one that is even.

15.7.3.7 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 (9.8.1) instead. Specifically, perform the following steps:

1. Let *x* be thisNumberValue(**this** value).
2. ReturnIfAbrupt(*x*).
3. If *precision* is **undefined**, return ToString(*x*).
4. Let *p* be ToInteger(*precision*).
5. ReturnIfAbrupt(*p*).
6. If *x* is **NaN**, return the String **"NaN"**.
7. Let *s* be the empty String.
8. If $x < 0$, then
    a. Let *s* be **"-"**.
    b. Let *x* = –*x*.
9. If *x* = +∞, then
    a. Return the concatenation of the Strings *s* and **"Infinity"**.
10. If $p < 1$ or $p > 21$, throw a **RangeError** exception.
11. If *x* = 0, then
    a. Let *m* be the String consisting of *p* occurrences of the code unit 0x0030 (the Unicode character '0').
    b. Let *e* = 0.
12. Else *x* ≠ 0,
    a. Let *e* and *n* be integers such that $10^{p-1} \le n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-p+1}$ is larger.
    b. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
    c. If $e < -6$ or $e \ge p$, then
        i. Let *a* be the first element of *m*, and let *b* be the remaining *p*–1 elements of *m*.
        ii. Let *m* be the concatenation of the three Strings *a*, **"."**, and *b*.
        iii. If *e* = 0, then

Deleted:

Deleted: v

Deleted: character

Deleted: character

Deleted: characters

1. Let c = "+" and d = "0".

iv. Else $e \neq 0$,

1. If $e > 0$, then

a Let c = "+".

2. Else $e < 0$,

a Let c = "-".

b Let $e = -e$.

3. Let d be the String consisting of the digits of the decimal representation of e (in order, with no leading zeroes).

v. Let m be the concatenation of the five Strings s, m, "e", c, and d.

13. If $e = p-1$, then return the concatenation of the Strings s and m.
14. If $e \geq 0$, then
    a. Let m be the concatenation of the first e+1 elements of m, the code unit 0x002E (Unicode character '.'), and the remaining $p-(e+1)$ elements of m.
15. Else $e < 0$,
    a. Let m be the concatenation of the String "0.", $-(e+1)$ occurrences of code unit 0x0030 (the Unicode character '0'), and the String m.
16. Return the concatenation of the Strings s and m.

The **length** property of the **toPrecision** method is **1**.

If the **toPrecision** method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of **toPrecision** for values of *precision* less than 1 or greater than 21. In this case **toPrecision** would not necessarily throw **RangeError** for such values.

15.7.4.8 Number.prototype.clz ()

When the **Number.prototype.clz** is called with one argument *number*, the following steps are taken:

1. Let x be thisNumberValue(**this** value).
2. Let n be ToUint32(x).
3. ReturnIfAbrupt(n).
4. Let p be the number of leading zero bits in the 32-bit binary representation of n.
5. Return p.

NOTE If n is 0, p will be 32. If the most significant bit of the 32-bit binary encoding of n is 1, p will be 0.

15.7.4 Properties of Number Instances

Number instances are ordinary objects that inherit properties from the Number prototype object. Number instances also have a [[NumberData]] internal data property. The [[NumberData]] internal data property is the Number value represented by this Number object.

15.8 The Math Object

The Math object is a single ordinary object.

The value of the [[Prototype]] internal data property of the Math object is the standard built-in Object prototype object (15.2.4). The Math object has a [[MathTag]] internal data property whose value is **true**.

The Math is not a function object. It does not have a [[Construct]] internal method; it is not possible to use the Math object as a constructor with the **new** operator. The Math object also does not have a [[Call]] internal method; it is not possible to invoke the Math object as a function.

NOTE In this specification, the phrase "the Number value for x" has a technical meaning defined in 8.5.

Deleted: characters
Deleted: s
Deleted: characters
Deleted: c
Comment [AWB7177]: Added at March 29 TC39 meeting
Deleted: 3
Deleted: 14
Deleted:
Deleted: v
Comment [AWB7178]: Unsigned seems most gene signed values could be converted in a previous step.
Deleted: f
Deleted: and their have an [[ClassIsNumberNativeBrandBuiltinBrand]] internal dat property whose value is NumberWrapperBuiltinNumberWrapper
Deleted: value is "Number"
Deleted: PrimitiveValue
Deleted: Value
Deleted: ¶
Deleted: Value
Deleted: PrimitiveValue
Deleted: that has some named properties, some of which are functions
Deleted: value of the [[Class]] internal property of the
Deleted: is "Math"
Deleted: the
Deleted: IsMath
Deleted: NativeBrand
Deleted: BuiltinBrand
Deleted: NativeMath
Deleted: BuiltinMath
Deleted: object
Deleted: property
Deleted: ¶
Deleted: property

15.8.1 Value Properties of the Math Object

15.8.1.1 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** }.

15.8.1.2 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** }.

15.8.1.3 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** }.

15.8.1.4 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`.

15.8.1.5 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`.

15.8.1.6 Math.PI

The Number value for π, the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.8.1.7 Math.SQRT1_2

The Number value for the square root of ½, which is approximately 0.7071067811865476.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE The value of `Math.SQRT1_2` is approximately the reciprocal of the value of `Math.SQRT2`.

15.8.1.8 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** }.

15.8.2 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, functction performs a computation on the resulting Number value(s).

Deleted: operator

Deleted: and then

In the function descriptions below, the symbols NaN, −0, +0, −∞ and +∞ refer to the Number values described in 8.5.

NOTE The behaviour of the functions **`acos`**, **`asin`**, **`atan`**, **`atan2`**, **`cos`**, **`exp`**, **`log`**, **`pow`**, **`sin`**, **`sqrt`**, and **`tan`** is not precisely specified here except to require specific results for certain argument values that represent boundary cases of interest. For other argument values, these functions are intended to compute approximations to the results of familiar mathematical functions, but some latitude is allowed in the choice of approximation algorithms. The general intent is that an implementer should be able to use the same mathematical library for ECMAScript on a given hardware platform that is available to C programmers on that platform.

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in **`fdlibm`**, the freely distributable mathematical library from Sun Microsystems (http://www.netlib.org/fdlibm).

15.8.2.1 Math.abs (x)

Returns the absolute value of *x*; the result has the same magnitude as *x* but has positive sign.

- If *x* is NaN, the result is NaN.
- If *x* is −0, the result is +0.
- If *x* is −∞, the result is +∞.

15.8.2.2 Math.acos (x)

Returns an implementation-dependent approximation to the arc cosine of *x*. The result is expressed in radians and ranges from +0 to +π.

- If *x* is NaN, the result is NaN.
- If *x* is greater than 1, the result is NaN.
- If *x* is less than −1, the result is NaN.
- If *x* is exactly 1, the result is +0.

15.8.2.3 Math.asin (x)

Returns an implementation-dependent approximation to the arc sine of *x*. The result is expressed in radians and ranges from −π/2 to +π/2.

- If *x* is NaN, the result is NaN.
- If *x* is greater than 1, the result is NaN.
- If *x* is less than −1, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.

15.8.2.4 Math.atan (x)

Returns an implementation-dependent approximation to the arc tangent of *x*. The result is expressed in radians and ranges from −π/2 to +π/2.

- If *x* is NaN, the result is NaN.
- If *x* is +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 $-\pi/2$.

15.8.2.5 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 if 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 $-3\pi/4$.

15.8.2.6 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 `Math.ceil(x)` is the same as the value of `-Math.floor(-x)`.

15.8.2.7 Math.cos (x)

Returns an implementation-dependent approximation to the cosine of x. The argument is expressed in radians.

- If x is NaN, the result is NaN.
- If x is $+0$, the result is 1.
- If x is -0, the result is 1.
- If x is $+\infty$, the result is NaN.
- If x is $-\infty$, the result is NaN.

15.8.2.8 Math.exp (x)

Returns an implementation-dependent approximation to the exponential function of *x* (*e* raised to the power of *x*, where *e* is the base of the natural logarithms).

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is 1.
- If *x* is –0, the result is 1.
- If *x* is +∞, the result is +∞.
- If *x* is –∞, the result is +0.

15.8.2.9 Math.floor (x)

Returns the greatest (closest to **+**∞) Number value that is not greater than *x* and is equal to a mathematical integer. If *x* is already an integer, the result is *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞, the result is +∞.
- If *x* is –∞, the result is –∞.
- If *x* is greater than 0 but less than 1, the result is +0.

NOTE The value of `Math.floor(x)` is the same as the value of `-Math.ceil(-x)`.

15.8.2.10 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 +∞.

15.8.2.11 Math.max ([value1 [, value2 [, …]]])

Given zero or more arguments, calls ToNumber on each of the arguments and returns the largest of the resulting values.

- If no arguments are given, the result is –∞.
- If any value is NaN, the result is NaN.
- The comparison of values to determine the largest value is done using the Abstract Relational Comparison algorithm (11.8.1) except that +0 is considered to be larger than –0.

The `length` property of the `max` method is **2**.

Deleted: Comparision
Deleted: A
Deleted: og
Deleted: as in
Deleted: 5

15.8.2.12 Math.min ([value1 [, value2 [, …]]])

Given zero or more arguments, calls ToNumber on each of the arguments and returns the smallest of the resulting values.

- If no arguments are given, the result is +∞.
- If any value is NaN, the result is NaN.
- The comparison of values to determine the smallest value is done using the Abstract Relational Comparison algorithm (11.8.1) except that +0 is considered to be larger than –0.

The `length` property of the `min` method is **2**.

Deleted: Comparision
Deleted: A
Deleted: og
Deleted: as in
Deleted: 5

15.8.2.13 Math.pow (x, y)

Returns an implementation-dependent approximation to the result of raising x to the power y.

- If y is NaN, the result is NaN.
- If y is +0, the result is 1, even if x is NaN.
- If y is −0, the result is 1, even if x is NaN.
- If x is NaN and y is nonzero, the result is NaN.
- If abs(x)>1 and y is +∞, the result is +∞.
- If abs(x)>1 and y is −∞, the result is +0.
- If abs(x) is 1 and y is +∞, the result is NaN. Deleted:)==
- If abs(x) is 1 and y is −∞, the result is NaN. Deleted:)==
- If abs(x)<1 and y is +∞, the result is +0.
- If abs(x)<1 and y is −∞, the result is +∞.
- If x is +∞ and y>0, the result is +∞.
- If x is +∞ and y<0, the result is +0.
- If x is −∞ and y>0 and y is an odd integer, the result is −∞.
- If x is −∞ and y>0 and y is not an odd integer, the result is +∞.
- If x is −∞ and y<0 and y is an odd integer, the result is −0.
- If x is −∞ and y<0 and y is not an odd integer, the result is +0.
- If x is +0 and y>0, the result is +0.
- If x is +0 and y<0, the result is +∞.
- If x is −0 and y>0 and y is an odd integer, the result is −0.
- If x is −0 and y>0 and y is not an odd integer, the result is +0.
- If x is −0 and y<0 and y is an odd integer, the result is −∞.
- If x is −0 and y<0 and y is not an odd integer, the result is +∞.
- If x<0 and x is finite and y is finite and y is not an integer, the result is NaN.

15.8.2.14 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.

15.8.2.15 Math.round (x)

Returns the Number value that is closest to x and is equal to a mathematical integer. If two integer Number values are equally close to x, then the result is the Number value that is closer to +∞. If x is already an integer, the result is x.

- If x is NaN, the result is NaN.
- If x is +0, the result is +0.
- If x is −0, the result is −0.
- If x is +∞, the result is +∞.
- If x is −∞, the result is −∞.
- If x is greater than 0 but less than 0.5, the result is +0.
- If x is less than 0 but greater than or equal to -0.5, the result is −0.

NOTE 1 `Math.round(3.5)` returns 4, but `Math.round(–3.5)` returns –3.

NOTE 2 The value of `Math.round(x)` is the same as the value of `Math.floor(x+0.5)`, except when `x` is −0 or is less than 0 but greater than or equal to -0.5; for these cases `Math.round(x)` returns −0, but `Math.floor(x+0.5)` returns +0.

15.8.2.16 Math.sin (x)

Returns an implementation-dependent approximation to the sine of x. The argument is expressed in radians.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞ or –∞, the result is NaN.

15.8.2.17 Math.sqrt (x)

Returns an implementation-dependent approximation to the square root of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is less than 0, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞, the result is +∞.

15.8.2.18 Math.tan (x)

Returns an implementation-dependent approximation to the tangent of *x*. The argument is expressed in radians.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞ or –∞, the result is NaN.

15.8.2.19 Math.log10 (x)

Returns an implementation-dependent approximation to the base 10 logarithm of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is less than 0, the result is NaN.
- If *x* is +0, the result is –∞.
- If *x* is –0, the result is –∞.
- If *x* is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.20 Math.log2 (x)

Returns an implementation-dependent approximation to the base 2 logarithm of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is less than 0, the result is NaN.
- If *x* is +0, the result is –∞.
- If *x* is –0, the result is –∞.
- If *x* is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.21 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 +∞, the result is +∞.

15.8.2.22 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 NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.
- If *x* is +∞, the result is +∞.
- If *x* is −∞, the result is -1.

15.8.2.23 Math.cosh(x)

Returns an implementation-dependent approximation to the hyperbolic cosine of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is 1.
- If *x* is −0, the result is 1.
- If *x* is +∞, the result is +∞.
- If *x* is −∞, the result is +∞.

NOTE The value of cosh(x) is the same as *(exp(x) + exp(-x))/2*.

15.8.2.24 Math.sinh(x)

Returns an implementation-dependent approximation to the hyperbolic sine of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.
- If *x* is +∞, the result is +∞.
- If x is −∞, the result is −∞.

NOTE The value of cosh(x) is the same as *(exp(x) - exp(-x))/2*.

15.8.2.25 Math.tanh(x)

Returns an implementation-dependent approximation to the hyperbolic tangent of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.
- If *x* is +∞, the result is +1.
- If *x* is −∞, the result is -1.

NOTE The value of tanh(x) is the same as *(exp(x) - exp(-x))/(exp(x) + exp(-x))*.

15.8.2.26 Math.acosh(x)

Returns an implementation-dependent approximation to the inverse hyperbolic cosine of *x*.

- If *x* is NaN, the result is NaN.

- If x is less than 1, the result is NaN.
- If x is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.27 Math.asinh(x)

Returns an implementation-dependent approximation to the inverse hyperbolic sine of *x*.

- If *x* is NaN, the result is NaN.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞, the result is +∞.
- If x is –∞, the result is –∞.

15.8.2.28 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 –∞.
- If *x* is +1, the result is +∞.
- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.

15.8.2.29 Math.hypot(value1 , value2, value3 = 0)

Deleted: [

Deleted:]

Given two or three arguments, hypot returns an implementation-dependent approximation of the square root of the sum of squares of up to three arguments.

Deleted: its

- 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.

The length property of the `hypot` function is 2.

15.8.2.30 Math.trunc(x)

Comment [AWB7179]: Concensus at March29 TC3 meeting was to drop this function. See March29 meeti notes

Deleted: 15.8.2.30 hypot2(value1 , value2 [, value3 Given two or three arguments, hypot2 returns an implementation-dependent approximation of the sum o squares of its arguments. ¶
<#>If no arguments are given, 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 NaN, the result is NaN.¶
<#>If all arguments are either +0 or -0, the result i +0.¶

Returns the integral part of the number *x*, removing any fractional digits. If *x* is already an integer, the result is *x*.

- If *x* is NaN, the result is NaN.
- If *x* is –0, the result is –0.
- If *x* is +0, the result is +0.
- If *x* is +∞, the result is +∞.
- If x is –∞, the result is –∞.
- 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.

15.8.2.31 Math.sign(x)

Returns the sign of the x, indicating whether x is positive, negative or zero.

- If *x* is NaN, the result is NaN.
- If *x* is –0, the result is –0.

- If *x* is +0, the result is +0.
- If *x* is negative and not −0, the result is −1.
- If *x* is positive and not +0, the result is +1.

15.8.2.32 **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 +∞, the result is +∞.
- If *x* is −∞, the result is −∞.

15.8.2.33 **Math.imul(x, y)**

When the **`Math.imul`** is called with arguments *x* and *y* the following steps are taken:

1. Let *a* be ToUint32(*x*).
2. ReturnIfAbrupt(*a*).
3. Let *a* be ToUint32(*y*).
4. ReturnIfAbrupt(*b*).
5. Let *product* be ($a \times b$) *x* modulo modulo 2^{32}.
6. If *product* $\geq 2^{31}$, return *product* $- 2^{32}$, otherwise return *product*.

15.9 Date Objects

15.9.1 Overview of Date Objects and Definitions of Abstract Operations

Deleted: Operators

The following functions are abstract operations that operate on time values (defined in 15.9.1.1). Note that, in every case, if any argument to one of these functions is **NaN**, the result will be **NaN**.

15.9.1.1 **Time Values and Time Range**

A Date object contains a Number indicating a particular instant in time to within a millisecond. Such a Number is called a *time value*. A time value may also be **NaN**, indicating that the Date object does not represent a specific instant of time.

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. In time values leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript Number values can represent all integers from –9,007,199,254,740,992 to 9,007,199,254,740,992; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly –100,000,000 days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value **+0**.

15.9.1.2 **Day Number and Time within Day**

A given time value *t* belongs to day number

$$\mathrm{Day}(t) = \mathrm{floor}(t \,/\, \mathrm{msPerDay})$$

where the number of milliseconds per day is

$$\mathrm{msPerDay} = 86400000$$

The remainder is called the time within the day:

$$\text{TimeWithinDay}(t) = t \text{ modulo msPerDay}$$

15.9.1.3 **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

$$\begin{aligned}\text{DaysInYear}(y) \quad &= 365 \text{ if } (y \text{ modulo } 4) \neq 0\\ &= 366 \text{ if } (y \text{ modulo } 4) = 0 \text{ and } (y \text{ modulo } 100) \neq 0\\ &= 365 \text{ if } (y \text{ modulo } 100) = 0 \text{ and } (y \text{ modulo } 400) \neq 0\\ &= 366 \text{ if } (y \text{ modulo } 400) = 0\end{aligned}$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year y is given by:

$$\text{DayFromYear}(y) = 365 \times (y–1970) + \text{floor}((y–1969)/4) – \text{floor}((y–1901)/100) + \text{floor}((y–1601)/400)$$

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:

$$\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that TimeFromYear}(y) \leq t$$

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned}\text{InLeapYear}(t) \quad &= 0 \text{ if DaysInYear(YearFromTime}(t)) = 365\\ &= 1 \text{ if DaysInYear(YearFromTime}(t)) = 366\end{aligned}$$

15.9.1.4 **Month Number**

Months are identified by an integer in the range 0 to 11, inclusive. The mapping MonthFromTime(*t*) from a time value *t* to a month number is defined by:

MonthFromTime(t)	= 0	if	0	≤ DayWithinYear(t) < 31
	= 1	if	31	≤ DayWithinYear (t) < 59+InLeapYear(t)
	= 2	if	59+InLeapYear(t)	≤ DayWithinYear (t) < 90+InLeapYear(t)
	= 3	if	90+InLeapYear(t)	≤ DayWithinYear (t) < 120+InLeapYear(t)
	= 4	if	120+InLeapYear(t)	≤ DayWithinYear (t) < 151+InLeapYear(t)
	= 5	if	151+InLeapYear(t)	≤ DayWithinYear (t) < 181+InLeapYear(t)
	= 6	if	181+InLeapYear(t)	≤ DayWithinYear (t) < 212+InLeapYear(t)
	= 7	if	212+InLeapYear(t)	≤ DayWithinYear (t) < 243+InLeapYear(t)
	= 8	if	243+InLeapYear(t)	≤ DayWithinYear (t) < 273+InLeapYear(t)
	= 9	if	273+InLeapYear(t)	≤ DayWithinYear (t) < 304+InLeapYear(t)
	= 10	if	304+InLeapYear(t)	≤ DayWithinYear (t) < 334+InLeapYear(t)
	= 11	if	334+InLeapYear(t)	≤ DayWithinYear (t) < 365+InLeapYear(t)

where

$$\text{DayWithinYear}(t) = \text{Day}(t)–\text{DayFromYear(YearFromTime}(t))$$

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that MonthFromTime(0) = 0, corresponding to Thursday, 01 January, 1970.

15.9.1.5 **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 month number is defined by:

$$
\begin{array}{lll}
\mathrm{DateFromTime}(t) & = \mathrm{DayWithinYear}(t)+1 & \text{if } \mathrm{MonthFromTime}(t)=0 \\
 & = \mathrm{DayWithinYear}(t)-30 & \text{if } \mathrm{MonthFromTime}(t)=1 \\
 & = \mathrm{DayWithinYear}(t)-58-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=2 \\
 & = \mathrm{DayWithinYear}(t)-89-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=3 \\
 & = \mathrm{DayWithinYear}(t)-119-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=4 \\
 & = \mathrm{DayWithinYear}(t)-150-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=5 \\
 & = \mathrm{DayWithinYear}(t)-180-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=6 \\
 & = \mathrm{DayWithinYear}(t)-211-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=7 \\
 & = \mathrm{DayWithinYear}(t)-242-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=8 \\
 & = \mathrm{DayWithinYear}(t)-272-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=9 \\
 & = \mathrm{DayWithinYear}(t)-303-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=10 \\
 & = \mathrm{DayWithinYear}(t)-333-\mathrm{InLeapYear}(t) & \text{if } \mathrm{MonthFromTime}(t)=11
\end{array}
$$

15.9.1.6 Week Day

The weekday for a particular time value t is defined as

$$\mathrm{WeekDay}(t) \quad = (\mathrm{Day}(t) + 4) \text{ modulo } 7$$

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.

15.9.1.7 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.

15.9.1.8 15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to make its best effort to determine the local daylight saving time adjustment. An implementation dependent algorithm using best available information on time zones to determine the local daylight saving time adjustment DaylightSavingTA(t), measured in milliseconds.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

$$\mathrm{LocalTime}(t) = t + \mathrm{LocalTZA} + \mathrm{DaylightSavingTA}(t)$$

Conversion from local time to UTC is defined by

$$\mathrm{UTC}(t) = t - \mathrm{LocalTZA} - \mathrm{DaylightSavingTA}(t - \mathrm{LocalTZA})$$

Note that UTC(LocalTime(t)) is not necessarily always equal to t.

15.9.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

$$
\begin{array}{ll}
\mathrm{HourFromTime}(t) & = \mathrm{floor}(t / \mathrm{msPerHour}) \text{ modulo } \mathrm{HoursPerDay} \\
\mathrm{MinFromTime}(t) & = \mathrm{floor}(t / \mathrm{msPerMinute}) \text{ modulo } \mathrm{MinutesPerHour} \\
\mathrm{SecFromTime}(t) & = \mathrm{floor}(t / \mathrm{msPerSecond}) \text{ modulo } \mathrm{SecondsPerMinute} \\
\mathrm{msFromTime}(t) & = t \text{ modulo } \mathrm{msPerSecond}
\end{array}
$$

where

$$
\begin{array}{ll}
\mathrm{HoursPerDay} & = 24 \\
\mathrm{MinutesPerHour} & = 60
\end{array}
$$

Deleted: The value LocalTZA does not vary with time but depends only on the geographic location.

Comment [AWB8180]: Need a reference

Deleted: algorithm

Deleted: The

Deleted: ,

Deleted: must depend only on four things:¶
(1) the time since the beginning of the year¶
t – TimeFromYear(YearFromTime(t))¶
(2) whether t is in a leap year¶
InLeapYear(t)¶
(3) the week day of the beginning of the year
WeekDay(TimeFromYear(YearFromTime(t)))
and (4) the geographic location.¶
The implementation of ECMAScript should no
try to determine whether the exact time was
subject to daylight saving time, but just wheth
daylight saving time would have been in effe
the current daylight saving time algorithm had
been used at the time. This avoids
complications such as taking into account the
years that the locale observed daylight savin
time year round.¶
If the host environment provides functionality
determining daylight saving time, the
implementation of ECMAScript is free to map
the year in question to an equivalent year (sa
leap-year-ness and same starting week day f
the year) for which the host environment
provides daylight saving time information. Th
only restriction is that all equivalent years
should produce the same result.

SecondsPerMinute = 60

msPerSecond = 1000

msPerMinute = 60000 = msPerSecond × SecondsPerMinute

msPerHour = 3600000 = msPerMinute × MinutesPerHour

15.9.1.11 MakeTime (hour, min, sec, ms)

The operator MakeTime calculates a number of milliseconds from its four arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If *hour* is not finite or *min* is not finite or *sec* is not finite or *ms* is not finite, return **NaN**.
2. Let *h* be ToInteger(*hour*).
3. Let *m* be ToInteger(*min*).
4. Let *s* be ToInteger(*sec*).
5. Let *milli* be ToInteger(*ms*).
6. Let *t* be *h* ***** msPerHour **+** *m* ***** msPerMinute **+** *s* ***** msPerSecond **+** *milli*, performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators ***** and **+**).
7. Return *t*.

15.9.1.12 MakeDay (year, month, date)

The operator MakeDay calculates a number of days from its three arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If *year* is not finite or *month* is not finite or *date* is not finite, return **NaN**.
2. Let *y* be ToInteger(*year*).
3. Let *m* be ToInteger(*month*).
4. Let *dt* be ToInteger(*date*).
5. Let *ym* be *y* + floor(*m* /12).
6. Let *mn* be *m* modulo 12.
7. Find a value *t* such that YearFromTime(*t*) is *ym* and MonthFromTime(*t*) is *mn* and DateFromTime(*t*) is 1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Return Day(*t*) + *dt* − 1.

Deleted: ==

Deleted: ==

Deleted: ==

15.9.1.13 MakeDate (day, time)

The operator MakeDate calculates a number of milliseconds from its two arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If *day* is not finite or *time* is not finite, return **NaN**.
2. Return *day* × msPerDay + *time*.

15.9.1.14 TimeClip (time)

The operator TimeClip calculates a number of milliseconds from its argument, which must be an ECMAScript Number value. This operator functions as follows:

1. If *time* is not finite, return **NaN**.
2. If abs(*time*) > 8.64×10^{15}, return **NaN**.
3. Return ToInteger(*time*) + (**+0**). (Adding a positive zero converts **–0** to **+0**.)

Deleted: an implementation-dependent choice of either ToInteger(*time*) or

NOTE The point of step 3 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish **–0** and **+0**.

15.9.1.15 Date Time String Format

ECMAScript defines a string interchange format for date-times based upon a simplification of the ISO 8601 Extended Format. The format is as follows: **`YYYY-MM-DDTHH:mm:ss.sssZ`**

Where the fields are as follows:

`YYYY`	is the decimal digits of the year 0000 to 9999 in the Gregorian calendar.
`-`	"`-`" (hyphen) appears literally twice in the string.
`MM`	is the month of the year from 01 (January) to 12 (December).
`DD`	is the day of the month from 01 to 31.
`T`	"`T`" appears literally in the string, to indicate the beginning of the time element.
`HH`	is the number of complete hours that have passed since midnight as two decimal digits from 00 to 24.
`:`	"`:`" (colon) appears literally twice in the string.
`mm`	is the number of complete minutes since the start of the hour as two decimal digits from 00 to 59.
`ss`	is the number of complete seconds since the start of the minute as two decimal digits from 00 to 59.
`.`	"`.`" (dot) appears literally in the string.
`sss`	is the number of complete milliseconds since the start of the second as three decimal digits.
`Z`	is the time zone offset specified as "`Z`" (for UTC) or either "`+`" or "`-`" followed by a time expression **`HH:mm`**

This format includes date-only forms:

`YYYY`
`YYYY-MM`
`YYYY-MM-DD`

It also includes "date-time" forms that consist of one of the above date-only forms immediately followed by one of the following time forms with an optional time zone offset appended:

`THH:mm`
`THH:mm:ss`
`THH:mm:ss.sss`

All numbers must be base 10. If the **`MM`** or **`DD`** fields are absent "**`01`**" is used as the value. If the **`HH`**, **`mm`**, or **`ss`** fields are absent "**`00`**" is used as the value and the value of an absent **`sss`** field is "**`000`**". If the time zone offset is absent, the date-time is interpreted as a local time.

Deleted: The value of an
Deleted: absent
Deleted: in
Deleted: is "**`Z`**"

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.

15.9.1.15.1 Extended years

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,426 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the simplified ECMAScript format such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or – sign. The year 0 is considered positive and hence prefixed with a + sign.

NOTE Examples of extended years:

-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.
+287396-10-12T08:59:00.992Z	287396 A.D.

15.9.2 The Date Constructor

The Date constructor is the %Date% instrinsic object and the initial value of the **Date** property of the global object. When **Date** is called as a function rather than as a constructor, it returns a String representing the current time (UTC). However, if the **this** value passed in the call is an Object with an uninitialised [[DateValue]] internal data property, **Date** initialises the **this** object using the argument value. This permits **Date** to be used both as a function for creating data strings and to perform constructor instance initialization.

The **Date** constructor is designed to be subclassable. It may be used as the value of an **extends** clause of a class declaration. Subclass constructors that intended to inherit the specified **Date** behaviour must include a **super** call to the **Date** constructor to initialise the [[DateValue]] state of subclass instances.

15.9.2.1 Date (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

This description applies only if the Date constructor is called with at least two arguments.

When the **Date** function is called the following steps are taken:

1. Let *numberOfArgs* be the number of arguments passed to this constructor call.
2. Assert: *numberOfArgs* ≥ 2.
3. Let *O* be the **this** value.
4. If Type(*O*) is Object and *O* has a [[DateValue]] internal data property and the value of [[DateValue]] is **undefined**, then
   a. Let *y* be ToNumber(*year*).
   b. ReturnIfAbrupt(*year*).
   c. Let *m* be ToNumber(*month*).
   d. ReturnIfAbrupt(*month*).
   e. If *date* is supplied then let *dt* be ToNumber(*date*); else let *dt* be **1**.
   f. ReturnIfAbrupt(*dt*).
   g. If *hours* is supplied then let *h* be ToNumber(*hours*); else let *h* be **0**.
   h. ReturnIfAbrupt(*h*).
   i. If *minutes* is supplied then let *min* be ToNumber(*minutes*); else let *min* be **0**.
   j. ReturnIfAbrupt(*min*).
   k. If *seconds* is supplied then let *s* be ToNumber(*seconds*); else let *s* be **0**.
   l. ReturnIfAbrupt(*s*).
   m. If *ms* is supplied then let *milli* be ToNumber(*ms*); else let *milli* be **0**.
   n. ReturnIfAbrupt(*milli*).
   o. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+ToInteger(*y*); otherwise, let *yr* be *y*.
   p. Let *finalDate* be MakeDate(MakeDay(*yr*, *m*, *dt*), MakeTime(*h*, *min*, *s*, *milli*)).
   q. Set the [[DateValue]] internal data property of *O* to TimeClip(UTC(*finalDate*)).
   r. Return *O*.
5. Else,
   a. Return the result computed as if by the expression **(new Date()).toString()** where **Date** is this function and **toString** is the standard built-in method **Date.prototype.toString**.

Deleted: Called as a Function

Deleted: z

Deleted: ize

Deleted: **Date** may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

Deleted: ¶
NOTE The function call **Date(…)** is not equivalent to object creation expression **new Date(…)** with the sam arguments.¶
<#>15.9.2.1 Date ([year [, month [, date [, hou minutes [, seconds [, ms]]]]]]])¶
All of the arguments are optional; any arguments suppl are accepted but are completely ignored. A String is created and returned as if by the expression **(new Date()).toString()** where **Date** is the standard built-in constructor with that name and **toString** is th standard built-in method **Date.prototype.toStrin**
<#>15.9.3 The Date Constructor¶
When **Date** is called as part of a **new** expression, it is a constructor: it initialises the newly created ordinary obj
new

Deleted: When **Date** is called with two to seven arguments, it computes the date from *year*, *month*, and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*.¶

Deleted: The [[Prototype]] internal data property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of **Date.prototype** (15.9.4.1).¶
The [[Class]] internal property of the newly constructed object is set to **"Date"** has a [[IsDateNativeBrandBuiltinBrand]] internal data propert whose value is NativeDateBuiltinDate.¶
The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
The [[PrimitiveValueDateValue]] internal data property the newly constructed object is set as follows:¶

Deleted: PrimitiveValue

Deleted: the newly constructed object

Comment [AWB14181]: Need to express as an intrinsic from same realm as this function.

15.9.2.2 Date (value)

This description applies only if the Date constructor is called with exactly one argument.

When the `Date` function is called the following steps are taken:

1. Let *numberOfArgs* be the number of arguments passed to this constructor call.
2. Assert: *numberOfArgs* = 1.
3. Let *O* be the **this** value.
4. If Type(*O*) is Object and *O* has a [[DateValue]] internal data property and the value of [[DateValue]] is **undefined**, then
   a. If Type(*value*) is Object and *value* has a [[DateValue]] internal data property, 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 (15.9.4.2). If the parse resulted in an abrupt completion, *tv* is the Completion Record.
      iii. Else,
         1. Let *tv* be ToNumber(*v*).
   c. ReturnIfAbrupt(*tv*).
   d. Set the [[DateValue]] internal data property of *O* to TimeClip(*tv*).
   e. Return *O*.
5. Else,
   a. Return the result computed as if by the expression `(new Date()).toString()` where `Date` is this function and `toString` is the standard built-in method `Date.prototype.toString`.

15.9.2.3 Date ( )

This description applies only if the Date constructor is called with no arguments.

When the `Date` function is called the following steps are taken:

1. Let *numberOfArgs* be the number of arguments passed to this constructor call.
2. Assert: *numberOfArgs* = 0.
3. Let *O* be the **this** value.
4. If Type(*O*) is Object and *O* has a [[DateValue]] internal data property and the value of [[DateValue]] is **undefined**, then
   a. Set the [[DateValue]] internal data property of *O* to the time value (UTC) identifying the current time.
   b. Return *O*.
5. Else,
   a. Return the result computed as if by the expression `(new Date()).toString()` where `Date` is this function and `toString` is the standard built-in method `Date.prototype.toString`.

15.9.2.4 new Date (... argumentsList)

`Date` called as part of a new expression with argument list *argumentsList* it performs the following steps:

1. Let *F* be the `Date` function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If `Date` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

Deleted: new

Deleted: <#>The [[Prototype]] internal data property of the newly constructed object is set to the origin Date prototype object, the one that i the initial value of `Date.prototyp` (15.9.4.1).¶
<#>The newly constructed object h a [[IsDateNativeBrandBuiltinBrand]] internal data property whose value i NativeDateBuiltinDate.The [[Class]] internal property of the newly constructed object is set to `"Date"`.
<#>The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
<#>The [[PrimitiveValueDateValue]] internal data property of the newly constructed object is set as follows:¶

Deleted: *V*

Deleted: Parse

Deleted: ; let *V* be the time value f this date

Deleted: *return*

Deleted: *V*

Deleted: ,

Deleted: let

Deleted: *V*

Deleted: *V*

Deleted: Primitive

Deleted: the newly constructed object

Deleted: *V*

Deleted: <#>Return the newly constructed object.¶

Comment [AWB14182]: Need to express an intrinsic from same realm as this function.

Deleted: <#>Set the [[PrimitiveValue]] internal property of the newly construc object to TimeClip(*V*) and return.¶

Deleted: new

Comment [AWB14183]: Need to express an intrinsic from same realm as this function.

Deleted: The [[Prototype]] internal data property of the newly constructed object is se to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).¶
The newly constructed object has a [[IsDateNativeBrandBuiltinBrand]] internal da property whose value is NativeDateBuiltinDate.The [[Class]] internal property of the newly constructed object is se to `"Date"`.¶

Deleted: **<#>The Date Constructor¶**

Deleted: args

Deleted: args

Deleted: be the be the

Deleted: a

15.9.3 Properties of the Date Constructor

The value of the [[Prototype]] internal data property of the Date constructor is the Function prototype object (15.3.3).

Besides the **length** property (whose value is **7**), the Date constructor has the following properties:

Deleted: the internal properties and

15.9.3.1 Date.prototype

The initial value of **Date.prototype** is the built-in Date prototype object (15.9.5).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.9.3.2 Date.parse (string)

The **parse** function applies the ToString operator to its argument. If ToString results in an abrupt completion the Completion Record is immediately returned. Otherwise, **parse** interprets the resulting String as a date and time; it returns a Number, the UTC time value corresponding to the date and time. The String may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the String. The function first attempts to parse the format of the String according to the rules called out in Date Time String Format (15.9.1.15). If the String does not conform to that format the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Unrecognisable Strings or dates containing illegal element values in the format String shall cause **Date.parse** to return **NaN**.

Deleted:

Deleted: and

If *x* is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

```
x.valueOf()
Date.parse(x.toString())
Date.parse(x.toUTCString())
Date.parse(x.toISOString())
```

However, the expression

```
Date.parse(x.toLocaleString())
```

is not required to produce the same Number value as the preceding three expressions and, in general, the value produced by **Date.parse** is implementation-dependent when given any String value that does not conform to the Date Time String Format (15.9.1.15) and that could not be produced in that implementation by the **toString** or **toUTCString** method.

15.9.3.3 Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When the **UTC** function is called with fewer than two arguments, the behaviour is implementation-dependent. When the **UTC** function is called with two to seven arguments, it computes the date from *year*, *month* and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*. The following steps are taken:

1. Let *y* be ToNumber(*year*).
2. ReturnIfAbrupt(*y*).
3. Let *m* be ToNumber(*month*).
4. ReturnIfAbrupt(*m*).
5. If *date* is supplied then let *dt* be ToNumber(*date*); else let *dt* be **1**.
6. ReturnIfAbrupt(*dt*).
7. If *hours* is supplied then let *h* be ToNumber(*hours*); else let *h* be **0**.
8. ReturnIfAbrupt(*h*).
9. If *minutes* is supplied then let *min* be ToNumber(*minutes*); else let *min* be **0**.
10. ReturnIfAbrupt(*min*).
11. If *seconds* is supplied then let *s* be ToNumber(*seconds*); else let *s* be **0**.
12. ReturnIfAbrupt(*s*).
13. If *ms* is supplied then let *milli* be ToNumber(*ms*); else let *milli* be **0**.
14. ReturnIfAbrupt(*milli*).

Deleted: *s*

15. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+ToInteger(*y*); otherwise, let *yr* be *y*.
16. Return TimeClip(MakeDate(MakeDay(*yr*, *m*, *dt*), MakeTime(*h*, *min*, *s*, *milli*))).

The `length` property of the `UTC` function is **7**.

NOTE The `UTC` function differs from the `Date` constructor in two ways: it returns a time value as a Number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

15.9.3.4 Date.now ( )

The `now` function return a Number value that is the time value designating the UTC date and time of the occurrence of the call to `now`.

15.9.3.5 Date[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%DatePrototype%"`, ([[DateValue]])).
2. ReturnIfAbrupt(*obj*).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[DateValue]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the Date constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.9.4 Properties of the Date Prototype Object

The Date prototype object is itself an ordinary object. It is not a Date instance and does not have a [[DateValue]] internal data property.

The value of the [[Prototype]] internal data property of the Date prototype object is the standard built-in Object prototype object (15.2.4).

Unless explicitly stated 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 data property that has been initialised to a time value.

The abstract operation thisTimeValue(*value*) performs the following steps:

1. If Type(*value*) is Object and *value* has a [[DateValue]] internal data property, then
   a. Let *n* be the Number that is the value of *value's* [[NumberData]] internal data property.
   b. If *n* is not **undefined**, then return *n*.
2. Throw a **TypeError** exception.

In following descriptions of functions that are properties of the Date prototype object, the phrase "this Date object" refers to the object that is the **this** value for the invocation of the function. The phrase "this time value" within the specification of a method refers to the result returned by calling the abstract operation thisTimeValue with the **this** value of the method invocation passed as the argument.

15.9.4.1 Date.prototype.constructor

The initial value of `Date.prototype.constructor` is the built-in `Date` constructor.

Deleted: When t

Deleted: Date

Deleted: ,

Deleted: are taken

Deleted: <#>Add a [[DateValue]] internal da property to *obj*.¶
<#>Add a [[NativeBrand]] internal data proper to *obj*.¶
<#>Set the [[NativeBrand]] internal data prope of *obj* to BuiltinDate.¶

Deleted: **false**

Deleted: z

Deleted: Date

Deleted: (its [[Class]] is `"Date"`)

Deleted: and has a [[IsDateNativeBrandBuiltinBrand]] internal da property whose value is NativeDateBuiltinDa

Deleted: whose

Deleted:

Deleted: Its [[PrimitiveValueDateValue]] is N

Deleted: z

Deleted: Unless explicitly noted otherwise, none of these functions are generic; a **TypeError** exception is thrown if the **this** valu is not an object for which the value of the [[Class]]with a [[IsDateNativeBrandDateValue internal data property is `"Date"`. Also, the phrase "this time value" refers to the Number value for the time represented by this Date object, that is, the value of the [[PrimitiveValueDateValue]] internal data property of this Date object.¶

15.9.4.2 Date.prototype.toString ( )

This function returns a String value. If this time value is NaN, the String value is `"Invalid Date"`, otherwise the contents of the String are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form.

Deleted: The

NOTE For any Date value *d* whose milliseconds amount is zero, the result of `Date.parse(`*d*`.toString())` is equal to *d*`.valueOf()`. See 15.9.4.2.

15.9.4.3 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.

15.9.4.4 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.

15.9.4.5 Date.prototype.toLocaleString ( )

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.

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.4.6 Date.prototype.toLocaleDateString ( )

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.

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.4.7 Date.prototype.toLocaleTimeString ( )

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.

NOTE The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.4.8 Date.prototype.valueOf ( )

The **`valueOf`** function returns a Number, which is this time value.

15.9.4.9 15.9.5.9 Date.prototype.getTime ( )

1. Return this time value.

15.9.4.10 Date.prototype.getFullYear ( )

1. Let *t* be this time value.

2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return YearFromTime(LocalTime(*t*)).

15.9.4.11 **Date.prototype.getUTCFullYear ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return YearFromTime(*t*).

15.9.4.12 **Date.prototype.getMonth ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MonthFromTime(LocalTime(*t*)).

15.9.4.13 **Date.prototype.getUTCMonth ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MonthFromTime(*t*).

15.9.4.14 **Date.prototype.getDate ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return DateFromTime(LocalTime(*t*)).

15.9.4.15 **Date.prototype.getUTCDate ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return DateFromTime(*t*).

15.9.4.16 **Date.prototype.getDay ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return WeekDay(LocalTime(*t*)).

15.9.4.17 **Date.prototype.getUTCDay ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return WeekDay(*t*).

15.9.4.18 **Date.prototype.getHours ( )**

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return HourFromTime(LocalTime(*t*)).

15.9.4.19 Date.prototype.getUTCHours ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return HourFromTime(*t*).

15.9.4.20 Date.prototype.getMinutes ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MinFromTime(LocalTime(*t*)).

15.9.4.21 Date.prototype.getUTCMinutes ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MinFromTime(*t*).

15.9.4.22 Date.prototype.getSeconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return SecFromTime(LocalTime(*t*)).

15.9.4.23 Date.prototype.getUTCSeconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return SecFromTime(*t*).

15.9.4.24 Date.prototype.getMilliseconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return msFromTime(LocalTime(*t*)).

15.9.4.25 Date.prototype.getUTCMilliseconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return msFromTime(*t*).

15.9.4.26 Date.prototype.getTimezoneOffset ( )

Returns the difference between local time and UTC time in minutes.

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return (*t* − LocalTime(*t*)) / msPerMinute.

15.9.4.27 Date.prototype.setTime (time)

1. Let *v* be TimeClip(ToNumber(*time*)).
2. ReturnIfAbrupt(*v*).
3. Set the [[DateValue]] internal data property of this Date object to *v*.
4. Return *v*.

Deleted: PrimitiveValue

15.9.4.28 Date.prototype.setMilliseconds (ms)

1. Let *t* be the result of LocalTime(this time value).
2. Let *time* be MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), ToNumber(*ms*)).
3. Let *u* be TimeClip(UTC(MakeDate(Day(*t*), *time*))).
4. Set the [[DateValue]] internal data property of this Date object to *u*.
5. Return *u*.

Deleted: PrimitiveValue

15.9.4.29 Date.prototype.setUTCMilliseconds (ms)

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *time* be MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), ToNumber(*ms*)).
4. Let *v* be TimeClip(MakeDate(Day(*t*), *time*)).
5. Set the [[DateValue]] internal data property of this Date object to *v*.
6. Return *v*.

Deleted: PrimitiveValue

15.9.4.30 Date.prototype.setSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value `getMilliseconds()`.

1. Let *t* be the result of LocalTime(this time value).
2. Let *s* be ToNumber(*sec*).
3. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
4. Let *date* be MakeDate(Day(*t*), MakeTime(HourFromTime(*t*), MinFromTime(*t*), *s*, *milli*)).
5. Let *u* be TimeClip(UTC(*date*)).
6. Set the [[DateValue]] internal data property of this Date object to *u*.
7. Return *u*.

Deleted: PrimitiveValue

The `length` property of the `setSeconds` method is **2**.

15.9.4.31 Date.prototype.setUTCSeconds (sec [, ms])

If *ms* is not specified, this behaves as if *ms* were specified with the value `getUTCMilliseconds()`.

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *s* be ToNumber(*sec*).
4. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
5. Let *date* be MakeDate(Day(*t*), MakeTime(HourFromTime(*t*), MinFromTime(*t*), *s*, *milli*)).
6. Let *v* be TimeClip(*date*).
7. Set the [[DateValue]] internal data property of this Date object to *v*.
8. Return *v*.

Deleted: PrimitiveValue

The `length` property of the `setUTCSeconds` method is **2**.

15.9.4.32 Date.prototype.setMinutes (min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value `getSeconds()`.

If *ms* is not specified, this behaves as if *ms* were specified with the value `getMilliseconds()`.

1. Let *t* be the result of LocalTime(this time value).
2. Let *m* be ToNumber(*min*).
3. If *sec* is not specified, then let *s* be SecFromTime(*t*); otherwise, let *s* be ToNumber(*sec*).
4. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
5. Let *date* be MakeDate(Day(*t*), MakeTime(HourFromTime(*t*), *m*, *s*, *milli*)).
6. Let *u* be TimeClip(UTC(*date*)).
7. Set the [[DateValue]] internal data property of this Date object to *u*.
8. Return *u*.

Deleted: PrimitiveValue

The **length** property of the **setMinutes** method is **3**.

15.9.4.33 Date.prototype.setUTCMinutes (min [, sec [, ms]])

If *sec* is not specified, this behaves as if *sec* were specified with the value **getUTCSeconds()**.

If *ms* is not specified, this function behaves as if *ms* were specified with the value return by **getUTCMilliseconds()**.

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *m* be ToNumber(*min*).
4. If *sec* is not specified, then let *s* be SecFromTime(*t*); otherwise, let *s* be ToNumber(*sec*).
5. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
6. Let *date* be MakeDate(Day(*t*), MakeTime(HourFromTime(*t*), *m*, *s*, *milli*)).
7. Let *v* be TimeClip(*date*).
8. Set the [[DateValue]] internal data property of this Date object to *v*.
9. Return *v*.

Deleted: PrimitiveValue

The **length** property of the **setUTCMinutes** method is **3**.

15.9.4.34 Date.prototype.setHours (hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value **getMinutes()**.

If *sec* is not specified, this behaves as if *sec* were specified with the value **getSeconds()**.

If *ms* is not specified, this behaves as if *ms* were specified with the value **getMilliseconds()**.

1. Let *t* be the result of LocalTime(this time value).
2. Let *h* be ToNumber(*hour*).
3. If *min* is not specified, then let *m* be MinFromTime(*t*); otherwise, let *m* be ToNumber(*min*).
4. If If *sec* is not specified, then let *s* be SecFromTime(*t*); otherwise, let *s* be ToNumber(*sec*).
5. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
6. Let *date* be MakeDate(Day(*t*), MakeTime(*h*, *m*, *s*, *milli*)).
7. Let *u* be TimeClip(UTC(*date*)).
8. Set the [[DateValue]] internal data property of this Date object to *u*.
9. Return *u*.

Deleted: PrimitiveValue

The **length** property of the **setHours** method is **4**.

15.9.4.35 Date.prototype.setUTCHours (hour [, min [, sec [, ms]]])

If *min* is not specified, this behaves as if *min* were specified with the value **getUTCMinutes()**.

If *sec* is not specified, this behaves as if *sec* were specified with the value **getUTCSeconds()**.

If *ms* is not specified, this behaves as if *ms* were specified with the value **getUTCMilliseconds()**.

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *h* be ToNumber(*hour*).
4. If *min* is not specified, then let *m* be MinFromTime(*t*); otherwise, let *m* be ToNumber(*min*).
5. If *sec* is not specified, then let *s* be SecFromTime(*t*); otherwise, let *s* be ToNumber(*sec*).
6. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
7. Let *newDate* be MakeDate(Day(*t*), MakeTime(*h*, *m*, *s*, *milli*)).
8. Let *v* be TimeClip(*newDate*).
9. Set the [[DateValue]] internal data property of this Date object to *v*.
10. Return *v*.

Deleted: PrimitiveValue

The **length** property of the **setUTCHours** method is **4**.

15.9.4.36 Date.prototype.setDate (date)

1. Let *t* be the result of LocalTime(this time value).
2. Let *dt* be ToNumber(*date*).
3. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), MonthFromTime(*t*), *dt*), TimeWithinDay(*t*)).
4. Let *u* be TimeClip(UTC(*newDate*)).
5. Set the [[DateValue]] internal data property of this Date object to *u*.
6. Return *u*.

Deleted: PrimitiveValue

15.9.4.37 Date.prototype.setUTCDate (date)

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *dt* be ToNumber(*date*).
4. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), MonthFromTime(*t*), *dt*), TimeWithinDay(*t*)).
5. Let *v* be TimeClip(*newDate*).
6. Set the [[DateValue]] internal data property of this Date object to *v*.
7. Return *v*.

Deleted: PrimitiveValue

15.9.4.38 Date.prototype.setMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value **getDate()**.

1. Let *t* be the result of LocalTime(this time value).
2. Let *m* be ToNumber(*month*).
3. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
4. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), *m*, *dt*), TimeWithinDay(*t*)).
5. Let *u* be TimeClip(UTC(*newDate*)).
6. Set the [[DateValue]] internal data property of this Date object to *u*.
7. Return *u*.

Deleted: PrimitiveValue

The **length** property of the **setMonth** method is **2**.

15.9.4.39 Date.prototype.setUTCMonth (month [, date])

If *date* is not specified, this behaves as if *date* were specified with the value **getUTCDate()**.

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. Let *m* be ToNumber(*month*).
4. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
5. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), *m*, *dt*), TimeWithinDay(*t*)).
6. Let *v* be TimeClip(*newDate*).
7. Set the [[DateValue]] internal data property of this Date object to *v*.
8. Return *v*.

Deleted: PrimitiveValue

The **length** property of the **setUTCMonth** method is **2**.

15.9.4.40 Date.prototype.setFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value **getMonth()**.

If *date* is not specified, this behaves as if *date* were specified with the value **getDate()**.

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Let *y* be ToNumber(*year*).
3. If *month* is not specified, then let *m* be MonthFromTime(*t*); otherwise, let *m* be ToNumber(*month*).
4. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
5. Let *newDate* be MakeDate(MakeDay(*y*, *m*, *dt*), TimeWithinDay(*t*)).
6. Let *u* be TimeClip(UTC(*newDate*)).
7. Set the [[DateValue]] internal data property of this Date object to *u*.
8. Return *u*.

Deleted: PrimitiveValue

The **length** property of the **setFullYear** method is **3**.

15.9.4.41 Date.prototype.setUTCFullYear (year [, month [, date]])

If *month* is not specified, this behaves as if *month* were specified with the value **getUTCMonth()**.

If *date* is not specified, this behaves as if *date* were specified with the value **getUTCDate()**.

1. Let *t* be this time value; but if this time value is **NaN**, let *t* be **+0**.
2. ReturnIfAbrupt(*t*).
3. Let *y* be ToNumber(*year*).
4. If *month* is not specified, then let *m* be MonthFromTime(*t*); otherwise, let *m* be ToNumber(*month*).
5. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
6. Let *newDate* be MakeDate(MakeDay(*y*, *m*, *dt*), TimeWithinDay(*t*)).
7. Let *v* be TimeClip(*newDate*).
8. Set the [[DateValue]] internal data property of this Date object to *v*.
9. Return *v*.

Deleted: PrimitiveValue

The **length** property of the **setUTCFullYear** method is **3**.

15.9.4.42 Date.prototype.toUTCString ( )

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent the Date 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 15.9.1.15. It is not essential that the chosen format be unambiguous or easily machine parsable. If an implementation does not have a preferred human-readable format it is recommended to use the format defined in 15.9.1.15 but with a space rather than a "T" used to separate the date and time elements.

15.9.4.43 Date.prototype.toISOString ( )

This function returns a String value represent the instance in time represented by this Date object. The format of the String is the Date Time string format defined in 15.9.1.15. All fields are present in the String. The time zone is always UTC, denoted by the suffix Z. If the time value of this object is not a finite Number a **RangeError** exception is thrown.

15.9.4.44 Date.prototype.toJSON (key)

This function provides a String representation of a Date object for use by **JSON.stringify** (15.12.3).

When the **toJSON** method is called with argument *key*, the following steps are taken:

1. Let *O* be the result of calling ToObject, giving it the **this** value as its argument.
2. Let *tv* be ToPrimitive(*O*, hint Number).
3. If *tv* is a Number and is not finite, return **null**.
4. Let *toISO* be the result of Get(*O*, **"toISOString"**).
5. ReturnIfAbrupt(*toISO*).
6. If IsCallable(*toISO*) is **false**, throw a **TypeError** exception.
7. Return the result of calling the [[Call]] internal method of *toISO* with *O* as *thisArgument* and an empty List as *argumentsList*.

NOTE 1 The argument is ignored.

NOTE 2 The **toJSON** function is intentionally generic; it does not require that its **this** value be a Date object. Therefore, it can be transferred to other kinds of objects for use as a method. However, it does require that any such object have a **toISOString** method. An object is free to use the argument *key* to filter its stringification.

15.9.4.45 Date.prototype [@@ToPrimitive] (hint)

This function is called by ECMAScript language operators to convert an object to a primitive value. The allowed values for *hint* are **"default"**, **"number"**, and **"string"**. Date objects, are unique among built-in ECMAScript object in that they treat **"default"** as being equivalent to **"string"**. All other built-in ECMAScript objects treat **"default"** as being equivalent to **"number"**.

When the **@@ToPrimitive** method is called with argument *hint*, the following steps are taken:

1. Let *O* be this Date object.
2. If Type(*O*) is not Object, then throw a **TypeError** exception.
3. If *hint* is the string value **"string"** or the string value **"default"** , then
   a. Let *tryFirst* be **"string"**.
4. Else if *hint* is the string value **"number"**, then
   a. Let *tryFirst* be **"number"**.
5. Else, throw a **TypeError** exception.
6. Return the result of OrdinaryToPrimitive(*O*,*tryFirst*).

15.9.5 Properties of Date Instances

Date instances are ordinary objects that inherit properties from the Date prototype object. Date instances also have a [[DateValue]] internal data property. The [[DateValue]] internal data property is the time value represented by this Date object.

15.10 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.

15.10.1 Patterns

The **RegExp** constructor applies the following grammar to the input pattern String. An error occurs if the grammar cannot interpret the String as an expansion of *Pattern*.

Syntax

Pattern **::**
  Disjunction

Disjunction **::**
  Alternative
  Alternative **|** *Disjunction*

Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: the **this** value
Deleted: argument l
Deleted: .
Deleted: the
Deleted:
Deleted:
Deleted:
Deleted:
Deleted:
Deleted:
Deleted: and their have a [[ClassIsDateNativeBrandBuiltinBrand]] intern whose value is NativeDateBuiltinDate
Deleted: property value is **"Date"**
Deleted: PrimitiveValue
Deleted: ¶
Deleted: PrimitiveValue

Alternative **::**
[empty]
Alternative Term

Term **::**
Assertion
Atom
Atom Quantifier

Assertion **::**
`^`
`$`
`\ b`
`\ B`
`( ? =` *Disjunction* **`)`**
`( ? !` *Disjunction* **`)`**

Quantifier **::**
QuantifierPrefix
QuantifierPrefix **`?`**

QuantifierPrefix **::**
`*`
`+`
`?`
`{` *DecimalDigits* **`}`**
`{` *DecimalDigits* **`, }`**
`{` *DecimalDigits* **`,`** *DecimalDigits* **`}`**

Atom **::**
PatternCharacter
`.`
`\` *AtomEscape*
CharacterClass
`(` *Disjunction* **`)`**
`( ? :` *Disjunction* **`)`**

PatternCharacter **::**
SourceCharacter **but not one of**
`^ $ \ . * + ? ( ) [ ] { } |`

AtomEscape **::**
DecimalEscape
CharacterEscape
CharacterClassEscape

CharacterEscape **::**
ControlEscape
`c` *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

ControlEscape **:: one of**
`f n r t v`

ControlLetter **:: one of**
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

IdentityEscape **::**
  SourceCharacter **but not** *IdentifierPart*
  <ZWJ>
  <ZWNJ>

DecimalEscape **::**
  DecimalIntegerLiteral [lookahead ∉ DecimalDigit]

CharacterClassEscape **::** **one of**
  d D s S w W

CharacterClass **::**
  [[lookahead ∉ {^}] *ClassRanges* **]**
  [^ *ClassRanges* **]**

ClassRanges **::**
  [empty]
  NonemptyClassRanges

NonemptyClassRanges **::**
  ClassAtom
  ClassAtom NonemptyClassRangesNoDash
  ClassAtom **-** *ClassAtom ClassRanges*

NonemptyClassRangesNoDash **::**
  ClassAtom
  ClassAtomNoDash NonemptyClassRangesNoDash
  ClassAtomNoDash **-** *ClassAtom ClassRanges*

ClassAtom **::**
  -
  ClassAtomNoDash

ClassAtomNoDash **::**
  SourceCharacter **but not one of \ or] or -**
  **** *ClassEscape*

ClassEscape **::**
  DecimalEscape
  b
  CharacterEscape
  CharacterClassEscape

15.10.2 Pattern Semantics

A regular expression pattern is converted into an internal procedure using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same. The internal procedure is used as the value of a RegExp object's [[RegExpMatcher]] internal data property.

Deleted: [[Match]]

15.10.2.1 Notation

The descriptions below use the following variables:

- *Input* is the String being matched by the regular expression pattern. 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 the *Input* String.

- *NcapturingParens* is the total number of left capturing parentheses (i.e. the total number of times the *Atom* **::** **(** *Disjunction* **)** production is expanded) in the pattern. A left capturing parenthesis is any **(** pattern character that is matched by the **(** terminal of the *Atom* **::** **(** *Disjunction* **)** production.
- *IgnoreCase* is **true** if the RegExp object's [[OriginalFlags]] internal data propert contains `"i"` and otherwise is **false** y.
- *Multiline* is **true** if the RegExp object's [[OriginalFlags]] internal data property contains `"m"` and otherwise is **false**.

Deleted: the setting of the

Deleted: `ignoreCase`

Deleted: the setting of the

Deleted: `multiline`

Furthermore, the descriptions below use the following internal data structures:

- A *CharSet* is a mathematical set of characters.
- A *State* is an ordered pair (*endIndex*, *captures*) where *endIndex* is an integer and *captures* is an internal array of *NcapturingParens* values. *States* are used to represent partial match states in the regular expression matching algorithms. The *endIndex* is one plus the index of the last input character matched so far by the pattern, while *captures* holds the results of capturing parentheses. The n^{th} element of *captures* is either a String that represents the value obtained by the n^{th} set of capturing parentheses or **undefined** if the n^{th} set of capturing parentheses hasn't been reached yet. Due to backtracking, many *States* may be in use at any time during the matching process.
- A *MatchResult* is either a *State* or the special token **failure** that indicates that the match failed.
- A *Continuation* procedure is an internal closure (i.e. an internal procedure with some arguments already bound to values) that takes one *State* argument and returns a *MatchResult* result. If an internal closure references variables 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 the input String, starting at the intermediate state given by its *State* argument. If the match succeeds, the *Continuation* returns the final *State* that it reached; if the match fails, the *Continuation* returns **failure**.
- A *Matcher* procedure is an internal closure that takes two arguments -- a *State* and a *Continuation* -- and returns a *MatchResult* result. A *Matcher* attempts to match a middle subpattern (specified by the closure's already-bound arguments) of the pattern against the input String, starting at the intermediate state given by its *State* argument. The *Continuation* argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new *State*, the *Matcher* then calls *Continuation* on that new *State* to test if the rest of the pattern can match as well. If it can, the *Matcher* returns the *State* returned by *Continuation*; if not, the *Matcher* may try different choices at its choice points, repeatedly calling *Continuation* until it either succeeds or all possibilities have been exhausted.
- An *AssertionTester* procedure is an internal closure that takes a *State* argument and returns a Boolean result. The assertion tester tests a specific condition (specified by the closure's already-bound arguments) against the current place in the input String and returns **true** if the condition matched or **false** if not.
- An *EscapeValue* is either a character or an integer. An *EscapeValue* is used to denote the interpretation of a *DecimalEscape* escape sequence: a character *ch* means that the escape sequence is interpreted as the character *ch*, while an integer *n* means that the escape sequence is interpreted as a backreference to the n^{th} set of capturing parentheses.

15.10.2.2 Pattern

The production *Pattern* **::** *Disjunction* evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal closure that takes two arguments, a String *str* and an integer *index*, and performs the following:
    1. Let *Input* be the given String *str*. This variable will be used throughout the algorithms in 15.10.2.
    2. Let *InputLength* be the length of *Input*. This variable will be used throughout the algorithms in 15.10.2.
    3. Let *c* be a Continuation that always returns its State argument as a successful MatchResult.

4. Let *cap* be an internal array of *NcapturingParens* **undefined** values, indexed 1 through *NcapturingParens*.
5. Let *x* be the State (*index*, *cap*).
6. Call *m*(*x*, *c*) and return its result.

NOTE A Pattern evaluates ("compiles") to an internal procedure value. **`RegExp.prototype.exec`** can then apply this procedure to a String and an offset within the String to determine whether the pattern would match starting at exactly that offset within the String, and, if it does match, what the values of the capturing parentheses would be. The algorithms in 15.10.2 are designed so that compiling a pattern may throw a **SyntaxError** exception; on the other hand, once the pattern is successfully compiled, applying its result internal procedure to find a match in a String cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

15.10.2.3 Disjunction

The production *Disjunction* **::** *Alternative* evaluates by evaluating *Alternative* to obtain a Matcher and returning that Matcher.

The production *Disjunction* **::** *Alternative* **|** *Disjunction* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher *m1*.
2. Evaluate *Disjunction* to obtain a Matcher *m2*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Call *m1*(*x*, *c*) and let *r* be its result.
    2. If *r* isn't **failure**, return *r*.
    3. Call *m2*(*x*, *c*) and return its result.

NOTE The **|** regular expression operator separates two alternatives. The pattern first tries to match the left *Alternative* (followed by the sequel of the regular expression); if it fails, it tries to match the right *Disjunction* (followed by the sequel of the regular expression). If the left *Alternative*, the right *Disjunction*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*. Any capturing parentheses inside a portion of the pattern skipped by **|** produce **undefined** values instead of Strings. Thus, for example,

```
/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]
```

15.10.2.4 Alternative

The production *Alternative* **::** [empty] evaluates by returning a Matcher that takes two arguments, a State *x* and a Continuation *c*, and returns the result of calling *c*(*x*).

The production *Alternative* **::** *Alternative Term* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher *m1*.
2. Evaluate *Term* to obtain a Matcher *m2*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Create a Continuation *d* that takes a State argument *y* and returns the result of calling *m2*(*y*, *c*).
    2. Call *m1*(*x*, *d*) and return its result.

NOTE Consecutive *Terms* try to simultaneously match consecutive portions of the input String. 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*.

15.10.2.5 Term

The production *Term* **::** *Assertion* evaluates by returning an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

1. Evaluate *Assertion* to obtain an AssertionTester *t*.
2. Call *t*(*x*) and let *r* be the resulting Boolean value.
3. If *r* is **false**, return **failure**.
4. Call *c*(*x*) and return its result.

The production *Term* **::** *Atom* evaluates by evaluating *Atom* to obtain a Matcher and returning that Matcher.

The production *Term* **::** *Atom Quantifier* evaluates as follows:

1. Evaluate *Atom* to obtain a Matcher *m*.
2. Evaluate *Quantifier* to obtain the three results: an integer *min*, an integer (or ∞) *max*, and Boolean *greedy*.
3. If *max* is finite and less than *min*, then throw a **SyntaxError** exception.
4. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's *Term*. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this production's *Term* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *Term*.
5. Let *parenCount* be the number of left capturing parentheses in the expansion of this production's *Atom*. This is the total number of *Atom* :: **(** *Disjunction* **)** productions enclosed by this production's *Atom*.
6. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Call RepeatMatcher(*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*) and return its result.

Runtime Semantics: RepeatMatcher Abstract Operation

The abstract operation RepeatMatcher takes eight parameters, a Matcher *m*, an integer *min*, an integer (or ∞) *max*, a Boolean *greedy*, a State *x*, a Continuation *c*, an integer *parenIndex*, and an integer *parenCount*, and performs the following:

1. If *max* is zero, then call *c*(*x*) and return its result.
2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
    1. If *min* is zero and *y*'s *endIndex* is equal to *x*'s *endIndex*, then return **failure**.
    2. If *min* is zero then let *min2* be zero; otherwise let *min2* be *min*–1.
    3. If *max* is ∞, then let *max2* be ∞; otherwise let *max2* be *max*–1.
    4. Call RepeatMatcher(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*) and return its result.
3. Let *cap* be a fresh copy of *x*'s *captures* internal array.
4. For every integer *k* that satisfies *parenIndex* < *k* and *k* ≤ *parenIndex*+*parenCount*, set *cap*[*k*] to **undefined**.
5. Let *e* be *x*'s *endIndex*.
6. Let *xr* be the State (*e*, *cap*).
7. If *min* is not zero, then call *m*(*xr*, *d*) and return its result.
8. If *greedy* is **false**, then
    a. Call *c*(*x*) and let *z* be its result.
    b. If *z* is not **failure**, return *z*.
    c. Call *m*(*xr*, *d*) and return its result.
9. Call *m*(*xr*, *d*) and let *z* be its result.
10. If *z* is not **failure**, return *z*.
11. Call *c*(*x*) and return its result.

NOTE 1 An *Atom* followed by a *Quantifier* is repeated the number of times specified by the *Quantifier*. A *Quantifier* can be non-greedy, in which case the *Atom* pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the *Atom* pattern is repeated as many times as possible while still matching the sequel. The *Atom* pattern is repeated rather than the input String 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 $(n–1)^{st}$ repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the $(n-1)^{st}$ repetition of *Atom* and so on.

Compare

```
/a[a-z]{2,4}/.exec("abcdefghi")
```

which returns **`"abcde"`** with

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

which returns **`"abc"`**.

Consider also

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

which, by the choice point ordering above, returns the array

```
["aaba", "ba"]
```

and not any of:

```
["aabaac", "aabaac"]
["aabaac", "c"]
```

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaa,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

```
/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

which returns the array

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost **`*`** clears all captured Strings contained in the quantified *Atom*, which in this case includes capture Strings numbered 2, 3, 4, and 5.

NOTE 4 Step 1 of the RepeatMatcher's *d* closure states that, once the minimum number of repetitions has been satisfied, any more expansions of *Atom* that match the empty String are not considered for further repetitions. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```
/(a*)*/.exec("b")
```

or the slightly more complicated:

```
/(a*)b\1+/.exec("baaaac")
```

which returns the array

```
["b", ""]
```

15.10.2.6 Assertion

The production *Assertion* **:: `^`** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is zero, return **true**.
3. If *Multiline* is **false**, return **false**.
4. If the character *Input*[*e*–1] is one of *LineTerminator*, return **true**.
5. Return **false**.

The production *Assertion* **:: `$`** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is equal to *InputLength*, return **true**.
3. If *Multiline* is **false**, return **false**.
4. If the character *Input*[*e*] is one of *LineTerminator*, return **true**.
5. Return **false**.

Deleted: *multiline*

The production *Assertion* **::** `\ b` evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. Call *IsWordChar*(*e*–1) and let *a* be the Boolean result.
3. Call *IsWordChar*(*e*) and let *b* be the Boolean result.
4. If *a* is **true** and *b* is **false**, return **true**.
5. If *a* is **false** and *b* is **true**, return **true**.
6. Return **false**.

The production *Assertion* **::** `\ B` evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. Call *IsWordChar*(*e*–1) and let *a* be the Boolean result.
3. Call *IsWordChar*(*e*) and let b be the Boolean result.
4. If a is **true** and b is **false**, return **false**.
5. If a is **false** and b is **true**, return **false**.
6. Return **true**.

The production *Assertion* **::** `( ? =` *Disjunction* `)` evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
    1. Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
    2. Call *m*(*x*, *d*) and let *r* be its result.
    3. If *r* is **failure**, return **failure**.
    4. Let *y* be *r*'s State.
    5. Let *cap* be *y*'s *captures* internal array.
    6. Let *xe* be *x*'s *endIndex*.
    7. Let *z* be the State (*xe*, *cap*).
    8. Call *c*(*z*) and return its result.

The production *Assertion* **::** `( ? !` *Disjunction* `)` evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
    1. Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
    2. Call *m*(*x*, *d*) and let *r* be its result.
    3. If *r* isn't **failure**, return **failure**.
    4. Call *c*(*x*) and return its result.

Runtime Semantics: IsWordChar Abstract Operation

The abstract operation *IsWordChar* takes an integer parameter *e* and performs the following:

1. If *e* is –1 or *e* is *InputLength*, return **false**.
2. Let *c* be the character *Input*[*e*].
3. If *c* is one of the sixty-three characters below, return **true**.

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

Deleted: ==

Deleted: =

Deleted: =

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 _

4. Return **false**.

15.10.2.7 Quantifier

The production *Quantifier* :: *QuantifierPrefix* evaluates as follows:

1. Evaluate *QuantifierPrefix* to obtain the two results: an integer *min* and an integer (or ∞) *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 ∞) *max*.
2. Return the three results *min*, *max*, and **false**.

The production *QuantifierPrefix* **::** ***** evaluates by returning the two results 0 and ∞.

The production *QuantifierPrefix* **::** **+** evaluates by returning the two results 1 and ∞.

The production *QuantifierPrefix* **::** **?** evaluates by returning the two results 0 and 1.

The production *QuantifierPrefix* **::** **{** *DecimalDigits* **}** evaluates as follows:

1. Let *i* be the MV of *DecimalDigits* (see 7.8.3).
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 ∞.

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*.

15.10.2.8 Atom

The production *Atom* **::** *PatternCharacter* evaluates as follows:

1. Let *ch* be the character represented by *PatternCharacter*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* **::** **.** evaluates as follows:

1. Let *A* be the set of all characters except *LineTerminator*.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* **::** **** *AtomEscape* evaluates by evaluating *AtomEscape* to obtain a Matcher and returning that Matcher.

The production *Atom* **::** *CharacterClass* evaluates as follows:

1. Evaluate *CharacterClass* to obtain a CharSet *A* and a Boolean *invert*.
2. Call *CharacterSetMatcher*(*A*, *invert*) and return its Matcher result.

The production *Atom* :: **(** *Disjunction* **)** evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the *Atom* :: **(** *Disjunction* **)** production is expanded prior to this production's *Atom* plus the total number of *Atom* :: **(** *Disjunction* **)** productions enclosing this *Atom*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
    1. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following steps:
        1. Let *cap* be a fresh copy of *y*'s *captures* internal array.
        2. Let *xe* be *x*'s *endIndex*.
        3. Let *ye* be *y*'s *endIndex*.
        4. Let *s* be a fresh String whose characters are the characters of *Input* at positions *xe* (inclusive) through *ye* (exclusive).
        5. Set *cap*[*parenIndex*+1] to *s*.
        6. Let *z* be the State (*ye*, *cap*).
        7. Call *c*(*z*) and return its result.
    2. Call *m*(*x*, *d*) and return its result.

The production *Atom* :: **(? :** *Disjunction* **)** evaluates by evaluating *Disjunction* to obtain a Matcher and returning that Matcher.

Runtime Semantics: CharacterSetMatcher Abstract Operation

The abstract operation *CharacterSetMatcher* takes two arguments, a CharSet *A* and a Boolean flag *invert*, and performs the following:

1. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
    1. Let *e* be *x*'s *endIndex*.
    2. If *e* is *InputLength*, return **failure**.
    3. Let *ch* be the character *Input*[*e*].
    4. Let *cc* be the result of *Canonicalize*(*ch*).
    5. If *invert* is **false**, then
        a. If there does not exist a member *a* of set *A* such that *Canonicalize*(*a*) is *cc*, return **failure**.
    6. Else *invert* is **true**,
        a. If there exists a member *a* of set *A* such that *Canonicalize*(*a*) is *cc*, return **failure.**
    7. Let *cap* be *x*'s *captures* internal array.
    8. Let *y* be the State (*e*+1, *cap*).
    9. Call *c*(*y*) and return its result.

Deleted: ==

Deleted: ==

Deleted: ==

Runtime Semantics: Canonicalize Abstract Operation

The abstract operation *Canonicalize* takes a character parameter *ch* and performs the following steps:

1. If *IgnoreCase* is **false**, return *ch*.
2. Let *u* be *ch* converted to upper case as if by calling the standard built-in method `String.prototype.toUpperCase` on the one-character String *ch*.
3. If *u* does not consist of a single character, return *ch*.
4. Let *cu* be *u*'s character.
5. If *ch*'s code unit value is greater than or equal to decimal 128 and *cu*'s code unit value is less than decimal 128, then return *ch*.
6. Return *cu*.

NOTE 1 Parentheses of the form **(** *Disjunction* **)** serve both to group the components of the *Disjunction* pattern together and to save the result of the match. The result can be used either in a backreference (**** followed by a nonzero decimal number), referenced in a replace String, or returned as part of an array from the regular expression matching internal procedure. To inhibit the capturing behaviour of parentheses, use the form **(?:** *Disjunction* **)** instead.

NOTE 2 The form **(?=** *Disjunction* **)** specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside *Disjunction* must match at the current position, but the current position is not advanced before matching the sequel. If *Disjunction* can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a **(?=** form (this unusual behaviour is inherited from Perl). This only matters when the *Disjunction* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

```
/(?=(a+))/.exec("baaabac")
```

matches the empty String immediately after the first **b** and therefore returns the array:

```
["", "aaa"]
```

To illustrate the lack of backtracking into the lookahead, consider:

```
/(?=(a+))a*b\1/.exec("baaabac")
```

This expression returns

```
["aba", "a"]
```

and not:

```
["aaaba", "a"]
```

NOTE 3 The form **(?!** *Disjunction* **)** specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside *Disjunction* must fail to match at the current position. The current position is not advanced before matching the sequel. *Disjunction* can contain capturing parentheses, but backreferences to them only make sense from within *Disjunction* itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **undefined** because the negative lookahead must fail for the pattern to succeed. For example,

```
/(.*?)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"]
```

In case-insignificant matches all characters are implicitly converted to upper case immediately before they are compared. However, if converting a character to upper case would expand that character into more than one character (such as converting **"ß"** (\u00DF) into **"SS"**), then the character is left as-is instead. The character is also left as-is if it is not an ASCII character but converting it to upper case would make it into an ASCII character. This prevents Unicode characters such as \u0131 and \u017F from matching regular expressions such as **/[a-z]/i**, which are only intended to match ASCII letters. Furthermore, if these conversions were allowed, then **/[^\W]/i** would match each of **a**, **b**, …, **h**, but not **i** or **s**.

15.10.2.9 AtomEscape

The production *AtomEscape* **::** *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an EscapeValue *E*.
2. If *E* is a character, then
   a. Let *ch* be *E*'s character.
   b. Let *A* be a one-element CharSet containing the character *ch*.
   c. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.
3. *E* must be an integer. Let *n* be that integer.
4. If *n*=0 or *n*>*NcapturingParens* then throw a **SyntaxError** exception.
5. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
   1. Let *cap* be *x*'s *captures* internal array.
   2. Let *s* be *cap*[*n*].
   3. If *s* is **undefined**, then call *c*(*x*) and return its result.
   4. Let *e* be *x*'s *endIndex*.
   5. Let *len* be *s*'s length.

Deleted: *NCapturingParens*

6. Let *f* be *e*+*len*.
7. If *f*>*InputLength*, return **failure**.
8. If there exists an integer *i* between 0 (inclusive) and *len* (exclusive) such that *Canonicalize*(*s*[*i*]) is not the same character as *Canonicalize*(*Input* [*e*+*i*]), then return **failure**.
9. Let *y* be the State (*f*, *cap*).
10. Call *c*(*y*) and return its result.

The production *AtomEscape* **::** *CharacterEscape* evaluates as follows:

1. Evaluate *CharacterEscape* to obtain a character *ch*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

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 (see 15.10.2.11). It is an error if the regular expression has fewer than *n* capturing parentheses. If the regular expression has *n* or more capturing parentheses but the *n*th one is **undefined** because it has not captured anything, then the backreference always succeeds.

15.10.2.10 CharacterEscape

The production *CharacterEscape* **::** *ControlEscape* evaluates by returning the character according to Table 35.

Table 35 — ControlEscape Character Values

ControlEscape	Code Unit	Name	Symbol
t	**\u0009**	horizontal tab	<HT>
n	**\u000A**	line feed (new line)	<LF>
v	**\u000B**	vertical tab	<VT>
f	**\u000C**	form feed	<FF>
r	**\u000D**	carriage return	<CR>

The production *CharacterEscape* **::** **c** *ControlLetter* evaluates as follows:

1. Let *ch* be the character represented by *ControlLetter*.
2. Let *i* be *ch*'s code unit value.
3. Let *j* be the remainder of dividing *i* by 32.
4. Return the character whose code unit value is *j*.

The production *CharacterEscape* **::** *HexEscapeSequence* evaluates by evaluating the CV of the *HexEscapeSequence* (see 7.8.6) and returning its character result.

Deleted: 4

The production *CharacterEscape* **::** *UnicodeEscapeSequence* evaluates by evaluating the CV of the *UnicodeEscapeSequence* (see 7.8.6) and returning its character result.

Deleted: 4

The production *CharacterEscape* **::** *IdentityEscape* evaluates by returning the character represented by *IdentityEscape*.

15.10.2.11 DecimalEscape

The production *DecimalEscape* **::** *DecimalIntegerLiteral* [lookahead ∉ *DecimalDigit*] evaluates as follows:

1. Let *i* be the MV of *DecimalIntegerLiteral*.

2. If *i* is zero, return the EscapeValue consisting of a <NUL> character (Unicode value 0000).
3. Return the EscapeValue consisting of the integer *i*.

The definition of "the MV of *DecimalIntegerLiteral*" is in 7.8.3.

NOTE If `\` is followed by a decimal number *n* whose first digit is not `0`, then the escape sequence is considered to be a backreference. It is an error if *n* is greater than the total number of left capturing parentheses in the entire regular expression. `\0` represents the <NUL> character and cannot be followed by a decimal digit.

15.10.2.12 **CharacterClassEscape**

The production *CharacterClassEscape* **::** `d` evaluates by returning the ten-element set of characters containing the characters `0` through `9` inclusive.

The production *CharacterClassEscape* **::** `D` evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **::** `d`.

The production *CharacterClassEscape* **::** `s` evaluates by returning the set of characters containing the characters that are on the right-hand side of the *WhiteSpace* (7.2) or *LineTerminator* (7.3) productions.

The production *CharacterClassEscape* **::** `S` evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **::** `s`.

The production *CharacterClassEscape* **::** `w` evaluates by returning the set of characters containing the sixty-three characters:

```
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 _
```

The production *CharacterClassEscape* **::** `W` evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* **::** `w`.

15.10.2.13 **CharacterClass**

The production *CharacterClass* **::** `[` [lookahead ∉ {`^`}] *ClassRanges* `]` evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the Boolean **false**.

The production *CharacterClass* **::** `[` `^` *ClassRanges* `]` evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the Boolean **true**.

15.10.2.14 **ClassRanges**

The production *ClassRanges* **::** [empty] evaluates by returning the empty CharSet.

The production *ClassRanges* **::** *NonemptyClassRanges* evaluates by evaluating *NonemptyClassRanges* to obtain a CharSet and returning that CharSet.

15.10.2.15 **NonemptyClassRanges**

The production *NonemptyClassRanges* **::** *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

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*.

Runtime Semantics: CharacterRange Abstract Operation

The abstract operation *CharacterRange* takes two CharSet parameters *A* and *B* and performs the following:

1. If *A* does not contain exactly one character or *B* does not contain exactly one character then throw a **SyntaxError** exception.
2. Let *a* be the one character in CharSet *A*.
3. Let *b* be the one character in CharSet *B*.
4. Let *i* be the code unit value of character *a*.
5. Let *j* be the code unit value of character *b*.
6. If *i* > *j* then throw a **SyntaxError** exception.
7. Return the set containing all characters numbered *i* through *j*, inclusive.

15.10.2.16 NonemptyClassRangesNoDash

The production *NonemptyClassRangesNoDash* **::** *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

The production *NonemptyClassRangesNoDash* **::** *ClassAtomNoDash NonemptyClassRangesNoDash* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *NonemptyClassRangesNoDash* to obtain a CharSet *B*.
3. Return the union of CharSets *A* and *B*.

The production *NonemptyClassRangesNoDash* **::** *ClassAtomNoDash* **-** *ClassAtom ClassRanges* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *ClassAtom* to obtain a CharSet *B*.
3. Evaluate *ClassRanges* to obtain a CharSet *C*.
4. Call *CharacterRange*(*A*, *B*) and let *D* be the resulting CharSet.
5. Return the union of CharSets *D* and *C*.

NOTE 1 *ClassRanges* can expand into single *ClassAtoms* and/or ranges of two *ClassAtoms* 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* code unit value is greater than the second *ClassAtom's* code unit 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 ASCII letters 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.

15.10.2.17 ClassAtom

The production *ClassAtom* **::** **-** evaluates by returning the CharSet containing the one character **-**.

The production *ClassAtom* **::** *ClassAtomNoDash* evaluates by evaluating *ClassAtomNoDash* to obtain a CharSet and returning that CharSet.

15.10.2.18 **ClassAtomNoDash**

The production *ClassAtomNoDash* **::** *SourceCharacter* **but not one of \ or] or –** evaluates by returning a one-element CharSet containing the character represented by *SourceCharacter*.

The production *ClassAtomNoDash* **::** **** *ClassEscape* evaluates by evaluating *ClassEscape* to obtain a CharSet and returning that CharSet.

15.10.2.19 **ClassEscape**

The production *ClassEscape* **::** *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then throw a **SyntaxError** exception.
3. Let *ch* be *E*'s character.
4. Return the one-element CharSet containing the character *ch*.

The production *ClassEscape* **::** **b** evaluates by returning the CharSet containing the one character <BS> (Unicode value 0008).

The production *ClassEscape* **::** *CharacterEscape* evaluates by evaluating *CharacterEscape* to obtain a character and returning a one-element CharSet containing that character.

The production *ClassEscape* **::** *CharacterClassEscape* evaluates by evaluating *CharacterClassEscape* to obtain a CharSet and returning that CharSet.

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.

15.10.3 The RegExp Constructor

Deleted: Called as a Function

The RegExp constructor is the %RegExp% instrinsic object and the initial value of the **RegExp** property of the global object. When **RegExp** is called as a function rather than as a constructor, it creates and initialises a new RegExp object. Thus the function call **RegExp(…)** is equivalent to the object creation expression **new RegExp(…)** with the same arguments. However, if the **this** value value passed in the call is an Object with an [[RegExpMatcher]] internal data property whose value is **undefined**, it initialises the **this** value using the argument values. This permits **RegExp** to be used both as factory method and to perform constructor instance initialization.

Deleted: ize

The **RegExp** constructor is designed to be subclassable. It may be used as the value of an **extends** clause of a class declaration. Subclass constructors that intended to inherit the specified **RegExp** behaviour must include a **super** call to the **RegExp** constructor to initialise subclass instances.

Deleted: **RegExp** may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

15.10.3.1 **RegExp(pattern, flags)**

The following steps are taken:

1. Let *func* be this **RegExp** function object.
2. Let *O* be the **this** value.
3. If Type(*O*) is not Object or Type(*O*) is Object and *O* does not have a [[RegExpMatcher]] internal data property or Type(*O*) is Object and *O* has a [[RegExpMatcher]] internal data property and the value of [[RegExpMatcher]] is not **undefined**, then
   a. If Type(*pattern*) is Object and *O* has a [[RegExpMatcher]] internal data property and *flags* is **undefined**, then
      i. Return *pattern*;
   b. Let *O* be the result of calling the abstract operation RegExpAlloc with argument *func*.
   c. ReturnIfAbrupt(*O*).
4. If Type(*pattern*) is Object and *pattern* has a [[RegExpMatcher]] internal data property, then

Comment [AWB14184]: This then clause corresponds to the "called as a function" case the ES5 spec.

a. If the value of *pattern's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
b. If *flags* is not **undefined**, then throw a **TypeError** exception.
c. Let *P* be the value of *pattern's* [[OriginalSource]] internal data property.
d. Let *F* be the value of *pattern's* [[OriginalFlags]] internal data property.

5. Else,
   a. let *P* be *pattern*.
   b. let *F* be *flags*.
6. Return the result of the abstract operation RegExpInitialise with arguments *O*, *P*, and *F*.

NOTE If pattern is supplied using a *StringLiteral*, the usual escape sequence substitutions are performed before the String is processed by RegExp. If pattern must contain an escape sequence to be recognised by RegExp, any backslash \ characters must be escaped within the *StringLiteral* to prevent them being removed when the contents of the *StringLiteral* are formed.

15.10.3.2 new RegExp(...argumentsList)

`RegExp` called as part of a new expression with argument list *argumentsList* it performs the following steps:

1. Let *F* be the **`RegExp`** function object on which the **`new`** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **`new`** operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If **`RegExp`** is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.10.3.3 Abstract Operations for the RegExp Constructor

Runtime Semantics: RegExpAlloc Abstract Operation

When the abstract operation RegExpAlloc with argument *constructor* is called, the following steps are taken:

1. Assert: *constructor* is a constructable function.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*constructor*, **`"%RegExpPrototype%"`**, ([[RegExpMatcher]], [[OriginalSource]], [[OriginalFlags]])).
3. Let *status* be the result of DefinePropertyOrThrow(*obj*, **`"lastIndex"`**, PropertyDescriptor {[[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}).
4. ReturnIfAbrupt(*status*).
5. Return *obj*.

NOTE [[RegExpMatcher]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the **RegExp** constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

Runtime Semantics: RegExpInitialise Abstract Operation

When the abstract operation RegExpInitialise with arguments *obj*, *pattern*, and *flags* is called, the following steps are taken:

1. If *pattern* is **undefined**, then let *P* be the empty String.
2. Else, let *P* be ToString(*pattern*).
3. ReturnIfAbrupt(*P*).
4. If *flags* is **undefined**, then let *F* be the empty String.
5. Else, let *F* be ToString(*flags*).
6. ReturnIfAbrupt(*F*).
7. If *F* contains any character other than **`"g"`**, **`"i"`**, **`"m"`**, **`"u"`**, or **`"y"`** or if it contains the same character more than once, then throw a **SyntaxError** exception.

Deleted: <#>If *pattern* is **undefined**, then let *P* be the empty String.¶
<#>Else, let *P* be ToString(*pattern*).¶
<#>ReturnIfAbrupt(*P*).¶
<#>If *flags* is **undefined**, then let *F* be the empty String
<#>Else, let *F* be ToString(*flags*).¶
<#>ReturnIfAbrupt(*F*).¶

Deleted: <#>If Type(*pattern*) is oObject and *pattern* has [[NativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp and *flags* is undefin then return *pattern*.¶

Deleted: ize

Deleted: *obj*

Deleted: the result of the abstraction operationabstract operation RegExpCreate with arguments *pattern* and *flags*

Deleted: i

Deleted: z

Deleted: Create

Deleted: ize

Deleted: The

Deleted: Create

Deleted: ize

Deleted: *P*

Deleted:

Deleted: *F*

Deleted: does the following:

Deleted: <#>Assert: *P* and *F* are both Strings.¶

8. Parse *P* interpreted as UTF-16 encoded Unicode characters using the grammars in 15.10.1 for the goal symbol *Pattern*. Throw a **SyntaxError** exception if *P* did not conform to the grammar or if all characters of *P* where not matched by the parse.
9. Set *obj's* [[RegExpMatcher]] internal data property to the internal procedure obtained by evaluating ("compiling") the step 3's parse of *P* and applying the semantics provided in 15.10.2.
10. Set the value of *obj's* [[OriginalSource]] internal data property to *P*.
11. Set the value of *obj's* [[OriginalFlags]] internal data property to *F*.
12. Let *putStatus* be the result of Put(*obj*, `"lastIndex"`, 0, **true**).
13. ReturnIfAbrupt(*putStatus*).
14. Return *obj*.

Runtime Semantics: RegExpCreate Abstract Operation

When the abstract operation RegExpCreate with arguments *P* and *F* is called, the following steps are taken:

1. Let *obj* be the result of calling the abstract operation RegExpAlloc with argument %RegExp%.
2. ReturnIfAbrupt(*obj*).
3. Return the result of the abstract operation RegExpInitialise with arguments *obj*, *P*, and *F* .

Runtime Semantics: EscapeRegExpPattern Abstract Operation

When the abstract operation EscapeRegExpPattern with arguments *P* and *F* is called, the following occurs:

Let *S* be a String in the form of a *Pattern* equivalent to *P* interpreted as UTF-16 encoded Unicode characters, in which certain characters are escaped as described below. *S* may or may not be identical to *P* or *pattern*; however, the internal procedure that would result from evaluating *S* as a *Pattern* must behave identically to the internal procedure given by the constructed object's [[RegExpMatcher]] internal data property. Separate calls to this abstract operation using the same values for *P* and *F* must produce identical results.

The characters `/` 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 `"(?:)"`.

Return *S*.

15.10.4 Properties of the RegExp Constructor

The value of the [[Prototype]] internal data property of the RegExp constructor is the standard built-in Function prototype object (15.3.3).

Besides the `length` property (whose value is **2**), the RegExp constructor has the following properties:

15.10.4.1 RegExp.prototype

The initial value of `RegExp.prototype` is the RegExp prototype object (15.10.6).

This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.10.4.2 RegExp[@@create] ( )

The @@create method of an object *F* performs the following:

1. Return the result of calling the abstract operation RegExpAlloc with argument *F*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

Deleted: <#>Let *escapedSource* be the result calling the abstract operation EscapeRegExpPatt with arguments *P* and *F*.¶

Deleted: Initialize

Deleted: <#>Assert: *P* and *F* are both Strings

Deleted: ize

Deleted: If *pattern* is an object *R* that has an [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp whose [[Class]] internal property is `"RegExp"` and *flags* is **undefined**, then let *P* be the pattern used to construct *R* and let *F* be the flags used to construct *R*. If *pattern* is an object *R* that has a [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp whose [[Class]] internal property is `"RegExp"` and *flags* is no **undefined**, then throw a **TypeError** exceptio Otherwise, let *P* be the empty String if *pattern* **undefined** and ToString(*pattern*) otherwise, a let *F* be the empty String if *flags* is **undefined** and ToString(*flags*) otherwise.¶
If the characters of *P* do not have the syntact form *Pattern*, then throw a **SyntaxError** exception. Otherwise let the newly constructe object have a [[Match]] internal data property obtained by evaluating ("compiling") the characters of *P* as a *Pattern* as described in 15.10.2. ¶
If *F* contains any character other than `"g"`, `"` or `"m"`, or if it contains the same character m than once, then throw a **SyntaxError** excepti
If a **SyntaxError** exception is not thrown, the

Deleted: [[Match]]

Deleted: o

Comment [AWB14185]: Why is this underspecified? Why not specify an required escaping? Do different implementation differ their results?

Deleted: The following properties of the new constructed object are data properties with th attributes that are specified in 15.10.7. The [[Value]] of each property is set as follows:¶
The `source` property of the newly constructe object is set to *S*.¶
The `global` property of the newly constructe object is set to a Boolean value that is **true** if contains the character `"g"` and **false** otherwi
The `ignoreCase` property of the newly constructed object is set to a Boolean value t is **true** if *F* contains the character `"i"` and **fa** otherwise.¶
The `multiline` property of the newly constructed object is set to a Boolean value t is **true** if *F* contains the character `"m"` and **fa** otherwise.¶
The `lastIndex` property of the newly constructed object is set to **0**.¶

Deleted: <#>The RegExp Constructor
When `RegExp` is called as part of a `new` expression, it is a constructor: it initialises the newly created object.¶

Deleted: the internal properties and

Deleted: false

15.10.5 Properties of the RegExp Prototype Object

The RegExp prototype object is an ordinary object. It is not a RegExp instance and does not have a [[RegExpMatcher]] internal data property or any of the other internal data properties of RegExp instance objects.

The value of the [[Prototype]] internal data property of the RegExp prototype object is the standard built-in Object prototype object (15.2.4).

The RegExp prototype object does not have a `valueOf` property of its own; however, it inherits the `valueOf` property from the Object prototype object.

15.10.5.1 RegExp.prototype.constructor

The initial value of `RegExp.prototype.constructor` is the standard built-in `RegExp` constructor.

15.10.5.2 RegExp.prototype.exec(string)

Performs a regular expression match of *string* against the regular expression and returns an Array object containing the results of the match, or **null** if *string* did not match.

The String ToString(*string*) is searched for an occurrence of the regular expression pattern as follows:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *R's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *S* be the value of ToString(*string*)
6. ReturnIfAbrupt(*S*).
7. Return the result of the RegExpExec abstract operation with arguments *R* and *S*.

Runtime Semantics: RegExpExec Abstract Operation

The abstract operation RegExpExec with arguments *R* (an object) and *S* (a string) performs the following steps:

1. Assert: *R* is an initialised RegExp instance.
2. Let *matcher* be the value of *R's* [[RegExpMatcher]] internal data property.
3. Let *length* be the length of *S*.
4. Let *lastIndex* be the result of Get(*R*, `"lastIndex"`).
5. Let *i* be the value of ToInteger(*lastIndex*).
6. ReturnIfAbrupt(*i*).
7. Let *global* be the result of ToBoolean(Get(*R*, `"global"`)).
8. ReturnIfAbrupt(*global*).
9. If *global* is **false**, then let *i* = 0.
10. Let *matchSucceeded* be **false**.
11. Repeat, while *matchSucceeded* is **false**
    a. If *i* < 0 or *i* > *length*, then
        i. Let *putStatus* be the result of Put(*R*, `"lastIndex"`, 0, **true**).
        ii. ReturnIfAbrupt(*putStatus*).
        iii. Return **null**.
    b. Let *r* be the result of calling the *matcher* with arguments *S* and *i*.
    c. If *r* is **failure**, then
        i. Let *i* = *i*+1.
    d. else
        i. Assert: *r* is a State.
        ii. Set *matchSucceeded* to **true**.
12. Let *e* be *r*'s *endIndex* value.

Deleted: The RegExp prototype object is itself a regu expression object; its has an [[IsRegExpNativeBrandBuiltinBrand]] internal data property whose value is NativeRegExpBuiltinRegExp [[Class]] is `"RegExp"`. The initial values of the RegExp prototype object's data properties (15.10.7) are set as i the object was created by the expression `new RegExp` where `RegExp` is that standard built-in constructor with that name.

Deleted: In the following descriptions of functions that are properties of the RegExp prototype object, the phra "this RegExp object" refers to the object that is the **this** value for the invocation of the function; a **TypeError** exception is thrown if the **this** value is not an object or object for which the value of thethat has an [[ClassIsRegExpNativeBrandBuiltinBrand]] internal dat property whose value is NativeRegExpBuiltinRegExp i not `"RegExp"`.¶

Deleted: TypeError

Deleted: <#>Let *R* be this RegExp object.¶ <#>ReturnIfAbrupt(*R*).¶

Deleted: abstraction operation

Deleted: z

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: calling the [[

Deleted:]] internal method of

Deleted:

Deleted: with argument

Deleted: calling Call the [[

Deleted:]] internal method of

Deleted:

Deleted: with arguments

Deleted: and

Deleted: Call

Deleted: [[Match]]

Deleted: internal method of

Deleted: *R*

Deleted: *[[Match]] returned*

Deleted: Let

Deleted: be the

Deleted: result of the call to [[Match]]

13. If *global* is **true**,
    a. Let *putStatus* be the result of Put(*R*, "`lastIndex`", *e*, **true**).
    b. ReturnIfAbrupt(*putStatus*).
14. Let *n* be the length of *r*'s *captures* array. (This is the same value as 15.10.2.1's *NcapturingParens*.)
15. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
16. Let *matchIndex* be *i*.
17. Assert: The following [[DefineOwnProperty]] calls will not result in an abrupt completion.
18. Call the [[DefineOwnProperty]] internal method of *A* with arguments "`index`" and Property Descriptor {[[Value]]: *matchIndex*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
19. Call the [[DefineOwnProperty]] internal method of *A* with arguments "`input`" and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
20. Call the [[DefineOwnProperty]] internal method of *A* with arguments "`length`" and Property Descriptor {[[Value]]: *n* + 1}.
21. Let *matchedSubstr* be the matched substring (i.e. the portion of *S* between offset *i* inclusive and offset *e* exclusive).
22. Call the [[DefineOwnProperty]] internal method of *A* with arguments "`0`" and Property Descriptor {[[Value]]: *matchedSubstr*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
23. For each integer *i* such that *i* > 0 and *i* ≤ *n*
    a. Let *captureI* be *i*th element of *r*'s *captures* array.
    b. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*i*) and Property Descriptor {[[Value]]: *captureI*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
24. Return *A*.

15.10.5.3 get RegExp.prototype.global

`RegExp.prototype.global` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
4. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
5. If *flags* is **undefined**, then throw a **TypeError** exception.
6. If *flags* contains the character "`g`", then return **true**.
7. Return **false**.

15.10.5.4 get RegExp.prototype.ignoreCase

`RegExp.prototype.ignoreCase` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
4. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
5. If *flags* is **undefined**, then throw a **TypeError** exception.
6. If *flags* contains the character "`i`", then return **true**.
7. Return **false**.

15.10.5.5 get RegExp.prototype.multiline

`RegExp.prototype.multiline` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
4. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
5. If *flags* is **undefined**, then throw a **TypeError** exception.

Deleted: calling Call the [[
Deleted:]] internal method of
Deleted:
Deleted: with arguments
Deleted: and
Deleted: *NCapturingParens*
Deleted: a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
Deleted: ,
Deleted: , and **true**
Deleted: ",
Deleted: , and **true**
Deleted: ",
Deleted: , and **true**
Deleted: ",
Deleted: , and **true**
Deleted:),
Deleted: , and **true**
Deleted: TypeError
Deleted: TypeError
Deleted: TypeError

6. If *flags* contains the character **"m"**, then return **true**.
7. Return **false**.

15.10.5.6 get RegExp.prototype.source

RegExp.prototype.source is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalSource]] internal data property throw a **TypeError** exception.
4. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
5. Let *src* be the value of *R's* [[OriginalSource]] internal data property.
6. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
7. If either *src* or *flags* is **undefined**, then throw a **TypeError** exception.
8. Return the result of the abstract operation EscapeRegExpPattern with arguments *src* and *flags*.

Deleted: TypeError

15.10.5.7 get RegExp.prototype.sticky

RegExp.prototype.sticky is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
4. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
5. If *flags* is **undefined**, then throw a **TypeError** exception.
6. If *flags* contains the character **"y"**, then return **true**.
7. Return **false**.

Deleted: .

Deleted: TypeError

15.10.5.8 RegExp.prototype.test(string)

The following steps are taken:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *R's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *S* be the value of ToString(*string*)
6. ReturnIfAbrupt(*S*).
7. Let *match* be the result of the RegExpExec abstract operation with arguments *R* and *S*.
8. ReturnIfAbrupt(*match*).
9. If *match* is not null, then return **true**; else return **false**.

Deleted: TypeError

Deleted: <#>Let *R* be this RegExp object.¶
<#>ReturnIfAbrupt(*R*).¶

Deleted: abstraction operation

Deleted: evaluating the `RegExp.prototype.exec` (15.10.6.2) algorithm upon this RegExp object using *string* the argument

15.10.5.9 get RegExp.prototype.unicode

RegExp.prototype.unicode is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[OriginalFlags]] internal data property throw a **TypeError** exception.
4. Let *flags* be the value of *R's* [[OriginalFlags]] internal data property.
5. If *flags* is **undefined**, then throw a **TypeError** exception.
6. If *flags* contains the character **"u"**, then return **true**.
7. Return **false**.

Deleted: TypeError

15.10.5.10 RegExp.prototype.toString()

1. Let *R* be the **this** value.
2. If Type(*R*) is not Object, then throw a **TypeError** exception.
3. If *R* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *R's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *pattern* be the result of ToString(Get(*R*, `"source"`)).
6. ReturnIfAbrupt(*pattern*).
7. Let *result* be the String value formed by concatenating `"/"`, *pattern*, and `"/"`.
8. Let *global* be the result of ToBoolean(Get(*R*, `"global"`)).
9. ReturnIfAbrupt(*global*).
10. If *global* is **true**, then append `"g"` as the last character of *result*.
11. Let *ignoreCase* be the result of ToBoolean(Get(*R*, `"ignoreCase"`)).
12. ReturnIfAbrupt(*ignoreCase*).
13. If *ignoreCase* is **true**, then append `"i"` as the last character of *result*.
14. Let *multiline* be the result of ToBoolean(Get(*R*, `"multiline"`)).
15. ReturnIfAbrupt(*multiline*).
16. If *multiline* is **true**, then append `"m"` as the last character of *result*.
17. Let *unicode* be the result of ToBoolean(Get(*R*, `"unicode"`)).
18. ReturnIfAbrupt(*unicode*).
19. If *unicode* is **true**, then append `"u"` as the last character of *result*.
20. Let *sticky* be the result of ToBoolean(Get(*R*, `"sticky"`)).
21. ReturnIfAbrupt(*sticky*).
22. If *sticky* is **true**, then append `"y"` as the last character of *result*.
23. Return *result*.

NOTE The returned String has the form of a *RegularExpressionLiteral* that evaluates to another RegExp object with the same behaviour as this object.

15.10.5.11 RegExp.prototype.match (string)

When the **match** method is called with argument *string*, the following steps are taken:

1. Let *rx* be the **this** value.
2. If Type(*rx*) is not Object, then throw a **TypeError** exception.
3. If *rx* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *rx's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *S* be the value of ToString(*string*)
6. ReturnIfAbrupt(*S*).
7. Let *global* be the result of ToBoolean(Get(*rx*, `"global"`)).
8. ReturnIfAbrupt(*global*).
9. If *global* is not **true**, then
   a. Return the result of calling the abstract operation RegExpExec (see 15.10.6.2) with arguments *rx* and *S*.
10. Else *global* is **true**,
    a. Let *putStatus* be the result of Put(*rx*, `"lastIndex"`, 0, **true**).
    b. ReturnIfAbrupt(*putStatus*).
    c. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
    d. Let *previousLastIndex* be 0.
    e. Let *n* be 0.
    f. Let *lastMatch* be **true**.
    g. Repeat, while *lastMatch* is **true**
       i. Let *result* be the result of the abstract operation RegExpExec with arguments *rx* and *S*.
       ii. ReturnIfAbrupt(*result*).
       iii. If *result* is **null**, then set *lastMatch* to **false**.
       iv. Else *result* is not **null**,
           1. Let *thisIndex* be the result of ToInteger(Get(*rx*, `"lastIndex"`)).

- **Deleted:** TypeError
- **Deleted:** the
- **Deleted:** *unicode*
- **Deleted:** `unicode`
- **Deleted:** *unicode*
- **Deleted:** SameValue(*unicode*, `"u"`)
- **Deleted:** `u`
- **Deleted:** SameValue(*ignoreCase*, `"i"`)
- **Deleted:** SameValue(*multiline*, `"m"`)
- **Deleted:** SameValue(*global*, `"y"`)
- **Deleted:** Return the String value formed by concatenating the Strings `"/"`, the String val of the **source** property of this RegExp object and `"/"`; plus `"g"` if the **global** property is **true**, `"i"` if the **ignoreCase** property is **true** and `"m"` if the **multiline** property is **true**.
- **Deleted:** TypeError

2. ReturnIfAbrupt(*thisIndex*).
3. If *thisIndex* = *previousLastIndex* then
    a Let *putStatus* be the result of Put(*rx*, `"lastIndex"`, *thisIndex*+1, **true**).
    b ReturnIfAbrupt(*putStatus*).
    c Set *previousLastIndex* to *thisIndex*+1.
4. Else,
    a Set *previousLastIndex* to *thisIndex*.
5. Let *matchStr* be the result of Get(*result*, `"0"`).
6. Let *defineStatus* be the result of DefinePropertyOrThrow(*A*, ToString(*n*), Property Descriptor {[[Value]]: *matchStr*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[configurable]]: **true**}).
7. ReturnIfAbrupt(*defineStatus*).
8. Increment *n*.

h. If *n* = 0, then return **null**.
i. Return *A*.

15.10.5.12 RegExp.prototype.replace (S, replaceValue)

When the `replace` method is called with arguments *S* and *replaceValue* the following steps are taken:

TODO: need tol finish this and have it make use of **GetReplaceSubstitution** operation in 15.5.4.11

1. Let *rx* be the **this** value.
2. If Type(*rx*) is not Object, then throw a **TypeError** exception.
3. If *rx* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *rx's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *string* be ToString(*S*).
6. ReturnIfAbrupt(*string*).
7. If *searchValue*.global is **false**, then search *string* for the first match of the regular expression *searchValue*. If *searchValue*.global is **true**, then search *string* for all matches of the regular expression *searchValue*. Do the search in the same manner as in `RegExp.prototype.match`, including the update of *searchValue*.`lastIndex`. Let *m* be the number of left capturing parentheses in *searchValue* (using *NcapturingParens* as specified in 15.10.2.1).
8. If *replaceValue* is a function, then
    a. For each matched substring, call the function with the following *m* + 3 arguments. Argument 1 is the substring that matched. If *searchValue* is a regular expression, the next *m* arguments are all of the captures in the MatchResult (see 15.10.2.1). Argument *m* + 2 is the offset within *string* where the match occurred, and argument *m* + 3 is *string*. The result is a String value derived from the original input by replacing each matched substring with the corresponding return value of the function call, converted to a String if need be.
9. Else,

Let *newstring* denote the result of converting *replaceValue* to a String. The result is a String value derived from the original input String by replacing each matched substring with a String derived from *newstring* by replacing elements in *newstring* by replacement text as specified in

a. Table 34. These `$` replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, `"$1,$2".replace(/(\$(\d))/g, "$$1-$1$2")` returns `"$1-$11,$1-$22"`. A `$` in *newstring* that does not match any of the forms below is left as is.

15.10.5.13 RegExp.prototype.search (S)

When the search method is called with argument *S*, the following steps are taken:

Deleted: TypeError

Deleted: Table 34

Comment [AWB7186]: Needs an real algorithm and also completion record checks.

1. Let *rx* be the **this** value.
2. If Type(*rx*) is not Object, then throw a **TypeError** exception.
3. If *rx* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *rx's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *string* be ToString(*S*).
6. ReturnIfAbrupt(*string*).
7. Search the value *string* from its beginning for an occurrence of the regular expression pattern *rx*. Let *result* be a Number indicating the offset within *string* where the pattern matched, or –1 if there was no match. If an abrupt completion occurs during the search, *result* is that Completion Record. The **lastIndex** and **global** properties of *regexp* are ignored when performing the search. The **lastIndex** property of *regexp* is left unchanged.
8. Return *result*.

Deleted: TypeError

Comment [AWB7187]: TODO: Needs bett spec langauge

15.10.5.14 RegExp.prototype.split (string, limit)

Returns an Array object into which substrings of the result of converting *string* to a String have been stored. The substrings are determined by searching from left to right for matches of the **this** value regular expression; these occurrences are not part of any substring in the returned array, but serve to divide up the String value.

The this value may be an empty regular expression or a regular expression that can match an empty String. In this case, regular expression does not match the empty substring at the beginning or end of the input String, nor does it match the empty substring at the end of the previous separator match. (For example, if the regular expression matches the empty String, the String is split up into individual code unit elements; the length of the result array equals the length of the String, and each substring contains one code unit.) Only the first match at a given position of the **this** String is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, `/a*?/.split("ab")` evaluates to the array `["a","b"]`, while `/a*/.split("ab")` evaluates to the array `["","b"]`.)

If the *string* is (or converts to) the empty String, the result depends on whether the regular expression can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

If the regular expression that contains capturing parentheses, then each time *separator* is matched the results (including any **undefined** results) of the capturing parentheses are spliced into the output array. For example,

```
/<(\/)?([^<>]+)>/.split("A<B>bold</B>and<CODE>coded</CODE>")
```

evaluates to the array

```
["A", undefined, "B", "bold", "/", "B", "and", undefined,
 "CODE", "coded", "/", "CODE", ""]
```

If *limit* is not **undefined**, then the output array is truncated so that it contains no more than *limit* elements.

When the `split` method is called, the following steps are taken:

1. Let *rx* be the **this** value.
2. If Type(*rx*) is not Object, then throw a **TypeError** exception.
3. If *rx* does not have a [[RegExpMatcher]] internal data property, then throw a **TypeError** exception.
4. If the value of *rx's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *matcher* be the value of *rx's* [[RegExpMatcher]] internal data property.
6. Let *S* be ToString(*string*).
7. ReturnIfAbrupt(*S*).
8. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
9. ReturnIfAbrupt(*A*).
10. Let *lengthA* be 0.
11. If *limit* is **undefined**, let *lim* = 2^{32}–1; else let *lim* = ToUint32(*limit*).
12. Let *s* be the number of elements in *S*.
13. Let *p* = 0.
14. If *lim* = 0, return *A*.

Deleted: TypeError

15. If *s* = 0, then
   a. Let *z* be the result of calling the *matcher* with arguments *S* and 0.
   b. ReturnIfAbrupt(z).
   c. If *z* is not **failure**, return *A*.
   d. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments `"0"` and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
   e. Assert: *status* is not an abrupt completion.
   f. Return *A*.
16. Let *q* = *p*.
17. Repeat, while *q* ≠ *s*
   a. Let *z* be the result of calling the *matcher* with arguments *S* and *q*
   b. ReturnIfAbrupt(z).
   c. If *z* is **failure**, then let *q* = *q*+1.
   d. Else *z* is not **failure**,
      i. *z* must be a State. Let *e* be *z*'s *endIndex* and let *cap* be *z*'s *captures* array.
      ii. If *e* = *p*, then let *q* = *q*+1.
      iii. Else *e* ≠ *p*,
         1. Let *T* be a String value equal to the substring of *S* consisting of the elements at positions *p* (inclusive) through *q* (exclusive).
         2. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
         3. Assert: *status* is not an abrupt completion.
         4. Increment *lengthA* by 1.
         5. If *lengthA* = *lim*, return *A*.
         6. Let *p* = *e*.
         7. Let *i* = 0.
         8. Repeat, while *i* is not equal to the number of elements in *cap*.
            a Let *i* = *i*+1.
            b Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *cap*[*i*], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            c Assert: *status* is not an abrupt completion.
            d Increment *lengthA* by 1.
            e If *lengthA* = *lim*, return *A*.
         9. Let *q* = *p*.
18. Let *T* be a String value equal to the substring of *S* consisting of the elements at positions *p* (inclusive) through *s* (exclusive).
19. Let *status* be the result of [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
20. Assert: *status* is not an abrupt completion.
21. Return *A*.

The **`length`** property of the **`split`** method is **2**.

NOTE 1 The **`split`** method ignores the value of the **`global`** property of this RegExp object.

15.10.5.15 RegExp.prototype.@@isRegExp

The initial value of the @@isRegExp property is **true**.

15.10.6 Properties of RegExp Instances

RegExp instances are ordinary objects that inherit properties from the RegExp prototype object. RegExp instances have internal data properties [[RegExpMatcher]], [[OriginalSource]], and [[OriginalFlags]]. The value of the [[RegExpMatcher]] internal data property is an implementation dependent representation of the *Pattern* of the RegExp object.

Deleted: q
Deleted: Call the
Deleted: the previous step will never result in
Deleted: Call the
Deleted: the previous step will never result in
Deleted: Call the
Deleted: the previous step will never result in
Deleted: Call the
Deleted: the previous step will never result in
Deleted: iterator
Deleted: and
Deleted: their
Deleted: a [[ClassIsRegExpNativeBrandBuiltinBrand]]
Deleted: property
Deleted: whose value is NativeRegExpBuiltinRegExp value is **`"RegExp"`**. RegExp instances also have a [[Match]]
Deleted: internal data property and a **`length`** proper
Deleted: ¶
Deleted: [[Match]]

NOTE Prior to the 6th Edition, **RegExp** instances were specified as having the own data properties **source**, **global**, **ignoreCase**, and **multiline**. Those properties are now specified as accessor properties of RegExp.prototype.

RegExp instances also have the following properties.

15.10.6.1 lastIndex

The value of the **lastIndex** property specifies the String position at which to start the next match. It is coerced to an integer when used (see 15.10.6.2). This property shall have the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE Unlike the other standard built-in properties of RegExp instances, **lastIndex** is writable.

15.11 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.

15.11.1 The Error Constructor

The Error constructor is the %Error% instrinsic object and the initial value of the **Error** property of the global object. When **Error** is called as a function rather than as a constructor, it creates and initialises a new Error object. Thus the function call **Error(…)** is equivalent to the object creation expression **new Error(…)** with the same arguments. However, if the **this** value value passed in the call is an Object with an uninitialised [[ErrorData]] internal data property, it initialises the **this** value using the argument value rather than creating a new object. This permits **Error** to be used both as factory method and to perform constructor instance initialization.

The **Error** constructor is designed to be subclassable. It may be used as the value of an **extends** clause of a class declaration. Subclass constructors that intended to inherit the specified **Error** behaviour must include a **super** call to the **Error** constructor to initialise subclass instances.

15.11.1.1 Error (message)

When the **Error** function is called with argument *message* the following steps are taken:

1. Let *func* be this **Error** function object.
2. Let *O* be the **this** value.
3. If Type(*O*) is not Object or Type(*O*) is Object and *O* does not have an [[ErrorData]] internal data property or Type(*O*) is Object and *O* has an [[ErrorData]] internal data property and the value of [[ErrorData]] is not **undefined**, then
   a. Let *O* be the result of calling OrdinaryCreateFromConstructor(*func*, **"%ErrorPrototype%"**, ([[ErrorData]]).
   b. ReturnIfAbrupt(*O*).
4. Assert: Type(*O*) is Object.
5. Set the value of *O's* [[ErrorData]] internal data property to any value other than **undefined**.
6. If *message* is not **undefined**, then
   a. Let *msg* be ToString(*message*).
   b. ReturnIfAbrupt(*msg*).
   c. Let *msgDesc* be the Property Descriptor {[[Value]]: *msg*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.
   d. Let *status* be the result of DefinePropertyOrThrow(*O*, **"message"**, *msgDesc*).
   e. ReturnIfAbrupt(*status*).
7. Return *O*.

Deleted: <#>15.10.7.1 source¶
The value of the **source** property is a String the form of a *Pattern* representing the current regular expression. This property shall have t attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.¶
<#>15.10.7.2 global¶
The value of the **global** property is a Boolea value indicating whether the flags contained t character **"g"**. This property shall have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.¶
<#>15.10.7.3 ignoreCase¶
The value of the **ignoreCase** property is a Boolean value indicating whether the flags contained the character **"i"**. This property sh have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }
<#>15.10.7.4 multiline¶
The value of the **multiline** property is a Boolean value indicating whether the flags contained the character **"m"**. This property sh have the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }

Deleted: Called as a Function

Deleted: generally

Deleted: z

Deleted: ize

Deleted: Error may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

Deleted: The

Comment [AWB14188]: This then clause corresponds to the "called as a function" case the ES5 spec.

Deleted: *F*

Deleted: <#>Else,¶

Deleted: <#>Let *status* be the result of CreateOwnDataProperty(*O*, "**message**", *msg*

Deleted: .

Deleted:

Deleted: calling the [[DefineOwnProperty]] internal method of

Deleted: passing

Deleted: and

Deleted:

Deleted: as arguments

Deleted: <#>If *status* is **false**, then throw a **TypeError** exception.¶

15.11.1.2 new Error(... argumentsList)

When **Error** called as part of a new expression with argument list *argumentsList* it performs the following steps:

1. Let *F* be the **Error** function object on which the **new** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **new** operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If **Error** is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.11.2 Properties of the Error Constructor

The value of the [[Prototype]] internal data property of the Error constructor is the Function prototype object (15.3.3).

Besides the internal properties and the **length** property (whose value is **1**), the Error constructor has the following property:

15.11.2.1 Error.prototype

The initial value of **Error.prototype** is the Error prototype object (15.11.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.11.2.2 Error[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, **"%ErrorPrototype%"**, ([[ErrorData]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[ErrorData]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the Boolean constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.11.3 Properties of the Error Prototype Object

The Error prototype object is an ordinary object. It is not an Error instance and does not have an [[ErrorData] internal data property.

The value of the [[Prototype]] internal data property of the Error prototype object is the standard built-in Object prototype object (15.2.4).

15.11.3.1 Error.prototype.constructor

The initial value of **Error.prototype.constructor** is the built-in **Error** constructor.

15.11.3.2 Error.prototype.name

The initial value of **Error.prototype.name** is **"Error"**.

Deleted: The [[Prototype]] internal data property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Error.prototype** (15.11.3.1).¶
The [[Class]] internal property of the newly constructed object is set to **"Error"** has an [[IsErrorNativeBrandBuiltinBrand]] internal data property whose value is NativeErrorBuiltinError.¶
The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
If the argument *message* is not **undefined**, the **message** own property of the newly constructed object is set to ToString(*message*). ¶

Deleted: <#>**The Error Constructor**¶
When **Error** called as part of a **new** expression, it is a constructor; it initialises the newly created object.¶

Deleted: **args**

Deleted: *args*

Deleted: be the be the

Deleted: a

Deleted: When **Error** is called as part of a **new** expression, it is a constructor: it initialises the newly created object.¶
<#>**15.11.2.1 new Error (message)**¶
The [[Prototype]] internal data property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Error.prototype** (15.11.3.1).¶
The [[Class]] internal property of the newly constructed Error object has an [[IsErrorNativeBrandBuiltinBrand]] internal data property whose value is NativeErrorBuiltinError is set to **"Error"**.¶
The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
If the argument *message* is not **undefined**, the **message** own property of the newly constructed object is set to ToString(*message*). ¶

Deleted: 15.3.4

Deleted: **false**

Deleted: z

Deleted: itself an Error object and has an [[IsErrorNativeBrandBuiltinBrand]] internal data property whose value is NativeErrorBuiltinError

Deleted: (its [[Class]] is **"Error"**)

15.11.3.3 **Error.prototype.message**

The initial value of **`Error.prototype.message`** is the empty String.

15.11.3.4 **Error.prototype.toString ( )**

The following steps are taken:

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a **TypeError** exception.
3. Let *name* be the result of Get(*O*, **`"name"`**).
4. ReturnIfAbrupt(*name*).
5. If *name* is **undefined**, then let *name* be **`"Error"`**; else let *name* be ToString(*name*).
6. Let *msg* be the result of Get(*O*, **`"message"`**).
7. ReturnIfAbrupt(*msg*).
8. If *msg* is **undefined**, then let *msg* be the empty String; else let *msg* be ToString(*msg*).
9. If *name* is the empty String, return *msg*.
10. If *msg* is the empty String, return *name*.
11. Return the result of concatenating *name*, **`":"`**, a single space character, and *msg*.

15.11.4 Properties of Error Instances

Error instances are ordinary objects that inherit properties from the Error prototype object and have a [[ErrorData]] internal data property whose initial value is **undefined**. The only specified uses of [[ErrorData]] is to flag whether or not an Error instance has been initialised by the Error constructor and to identify them as Error objects within **`Object.prototype.toString`**.

15.11.5 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 15.11.7.

15.11.5.1 **EvalError**

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

15.11.5.2 **RangeError**

Indicates a numeric value has exceeded the allowable range. See 15.4.2.2, 15.4.5.1, 15.7.4.2, 15.7.4.5, 15.7.4.6, 15.7.4.7, and 15.9.5.43.

15.11.5.3 **ReferenceError**

Indicate that an invalid reference value has been detected. See 8.9.1, 8.9.2, 10.2.1, 10.2.1.1.4, 10.2.1.2.4, and 11.13.1.

15.11.5.4 **SyntaxError**

Indicates that a parsing error has occurred. See 11.1.5, 11.3.1, 11.3.2, 11.4.1, 11.4.4, 11.4.5, 11.13.1, 11.13.2, 12.2.1, 12.10.1, 12.14.1, 13.1, 15.1.2.1, 15.3.2.1, 15.10.2.2, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, 15.10.4.1, and 15.12.2.

15.11.5.5 **TypeError**

Indicates the actual type of an operand is different than the expected type. See 8.6.2, 8.9.2, 8.10.5, 8.12.5, 8.12.7, 8.12.8, 8.12.9, 9.9, 9.10, 10.2.1, 10.2.1.1.3, 10.6, 11.2.2, 11.2.3, 11.4.1, 11.8.6, 11.8.7, 11.3.1, 13.2, 13.2.3, 15, 15.2.3.2, 15.2.3.3, 15.2.3.4, 15.2.3.5, 15.2.3.6, 15.2.3.7, 15.2.3.8, 15.2.3.9, 15.2.3.10, 15.2.3.11, 15.2.3.12, 15.2.3.13, 15.2.3.14, 15.2.4.3, 15.3.3.2, 15.3.3.3, 15.3.3.4, 15.3.3.5, 15.3.3.5.2, 15.3.3.5.3, 15.3.4,

Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Deleted: <#>If *msg* is undefined, then let *msg* the empty String; else let *msg* be ToString(*msg*).
Deleted: n
Deleted: IsError
Deleted: NativeBrand
Deleted: BuiltinBrand
Deleted: value is NativeErrorBuiltinError
Deleted: z
Deleted: .
Deleted: their [[Class]] internal property valu is **`"Error"`**
Deleted: . Error instances have no special properties.
Deleted: 7
Deleted: 7
Deleted: 7

15.3.4.3, 15.3.4.4, 15.4.3.3, 15.4.3.11, 15.4.3.16, 15.4.3.17, 15.4.3.18, 15.4.3.19, 15.4.3.20, 15.4.3.21, 15.4.3.22, 15.4.5.1, 15.5.4.2, 15.5.4.3, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.44, 15.10.4.1, 15.10.6, 15.11.4.4 and 15.12.3.

15.11.5.6 **URIError**

Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition. See 15.1.3.

15.11.6 *NativeError* Object Structure

When an ECMAScript implementation detects a runtime error, it throws an new instance of one of the *NativeError* objects defined in 15.11.5. 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 15.11.6.

15.11.6.1 *NativeError* Constructors

When a *NativeError* constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the object as a function is equivalent to calling it as a constructor with the same arguments. However, if the **this** value value passed in the call is an Object with an uninitialised [[ErrorData]] internal data property, it initialises the **this** value using the argument value. This permits **`Error`** to be used both as factory method and to perform constructor instance initialization.

The *NativeError* constructor is designed to be subclassable. It may be used as the value of an **`extends`** clause of a class declaration. Subclass constructors that intended to inherit the specified *NativeError* behaviour must include a **`super`** call to the *NativeError* constructor to initialise subclass instances.

15.11.6.1.1 *NativeError* (message)

When a *NativeError* function is called with argument *message* the following steps are taken:

1. Let *func* be this *NativeError* function object.
2. Let *O* be the **this** value.
3. If Type(*O*) is not Object or Type(*O*) is Object and *O* does not have an [[ErrorData]] internal data property or Type(*O*) is Object and *O* has an [[ErrorData]] internal data property and the value of [[ErrorData]] is not **undefined**, then
   a. Let *O* be the result of calling OrdinaryCreateFromConstructor(*func*, **`"%NativeErrorPrototype%"`**, ([[ErrorData]])).
   b. ReturnIfAbrupt(*O*).
4. Assert: Type(*O*) is Object.
5. Set the value of *O's* [[ErrorData]] internal data property to any value other than **undefined**.
6. If *message* is not **undefined**, then
   a. Let *msg* be ToString(*message*).
   b. Let *msgDesc* be the Property Descriptor {[[Value]]: *msg*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.
   c. Let *status* be the result of DefinePropertyOrThrow(*O*, **`"message"`**, *msgDesc*).
   d. ReturnIfAbrupt(*status*).
7. Return *O*.

The actual value of the string passed in step 3.a is either **`"%EvalErrorPrototype%"`**, **`"%RangeErrorPrototype%"`**, **`"%ReferenceErrorPrototype%"`**, **`"%SyntaxErrorPrototype%"`**, **`"%TypeErrorPrototype%"`**, or **`"%URIErrorPrototype%"`** corresponding to which *NativeError* constructor is being defined.

Comment [AWB10189]: Sectin references have not yet been updated to reflect ES6

Deleted: 15.11.6

Deleted: Called as Functions

Deleted: z

Deleted: ize

Deleted: A *NativeError* constructor may be subclassed and subclass constructors may perform a super invocation of

Deleted: ize

Comment [AWB14190]: This then clause corresponds to the "called as a function" case the ES5 spec.

Deleted: *F*

Deleted: <#>Else,¶

Deleted: <#>ReturnIfAbrupt(*msg*).¶
<#>Let *status* be the result of CreateOwnDataProperty(*O*, "**`message`**", *msg*).¶

Deleted: .

Deleted:

Deleted: calling the [[DefineOwnProperty]] internal method of

Deleted: passing

Deleted: and

Deleted: as arguments

Deleted: <#>If *status* is **false**, then throw a **TypeError** exception.¶

15.11.6.1.2 new *NativeError* (... argumentsList)

When a *NativeError* constructor is called as part of a new expression with argument list *argumentsList* it performs the following steps:

1. Let *F* be this *NativeError* function object on which the **new** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **new** operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If a *NativeError* constructor is invoked as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.11.6.2 Properties of the *NativeError* Constructors

The value of the [[Prototype]] internal data property of a *NativeError* constructor is the Error constructor object (15.11.1).

Besides the `length` property (whose value is **1**), each *NativeError* constructor has the following property:

15.11.6.2.1 *NativeError*.prototype

The initial value of ***NativeError*.prototype** is a *NativeError* prototype object (15.11.7.4). Each *NativeError* constructor has a separate prototype object.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.11.6.3 *NativeError* [@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, *NativeErrorPrototype*, ([[ErrorData]])).
3. Return *obj*.

The actual value passed as *NativeErrorPrototype* in step 2 is either `"%EvalErrorPrototype%"`, `"%RangeErrorPrototype%"`, `"%ReferenceErrorPrototype%"`, `"%SyntaxErrorPrototype%"`, `"%TypeErrorPrototype%"`, or `"%URIErrorPrototype%"` corresponding to which *NativeError* constructor is being defined.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE [[ErrorData]] is initially assigned the value **undefined** as a flag to indicate that the instance has not yet been initialised by the Boolean constructor. This flag value is never directly exposed to ECMAScript code; hence implementation may choose to encode the flag in some other manner.

15.11.6.4 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 data property.

The value of the [[Prototype]] internal data property of each *NativeError* prototype object is the standard built-in Error prototype object (15.11.4).

15.11.6.4.1 15.11.7.4.1 *NativeError*.prototype.constructor

The initial value of the `constructor` property of the prototype for a given *NativeError* constructor is the *NativeError* constructor function itself (15.11.7).

Deleted: The [[Prototype]] internal data property of the newly constructed object is se to the prototype object for this error construct The [[Class]] internal property of the newly constructed object has an [[IsErrorNativeBrandBuiltinBrand]] internal da property whose value is NativeErrorBuiltinErr is set to `"Error"`. The [[Extensible]] internal data property of the newly constructed object set to **true**.¶
If the argument *message* is not **undefined**, the `message` own property of the newly construc object is set to ToString(*message*).¶

Deleted: <#>The *NativeError* Constructors¶
When a *NativeError* constructor is called as p of a **new** expression, it is a constructor: it initialises the newly created object.¶

Deleted: args

Deleted: message

Formatted: Font: (Default) +Headings CS (Times New Roman), Complex Script Font: +Headings CS (Times New Roman)

Deleted: *args*

Deleted: be the be the

Deleted: a

Deleted: The [[Prototype]] internal data property of the newly constructed object is se to the prototype object for this *NativeError* constructor. The [[Class]] internal property o the newly constructed object has an [[IsErrorNativeBrandBuiltinBrand]] internal da property whose value is NativeErrorBuiltinErr is set to `"Error"`. The [[Extensible]] internal data property of the newly constructed object set to **true**.¶
If the argument *message* is not **undefined**, the `message` own property of the newly constructed object is set to ToString(*message*)

Deleted: Function

Deleted: prototype

Deleted: 3

Deleted: 4

Deleted: the internal properties and

Deleted: 7

Deleted: `"%ErrorPrototype%"`

Deleted: of the string

Deleted: false

Deleted: z

Deleted: an

Deleted: Error object and has an [[IsErrorNativeBrandBuiltinBrand]] internal da property whose value is NativeErrorBuiltinErr (its [[Class]] is `"Error"`).

Deleted: 8

Deleted: 6

15.11.6.4.2 ***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*).

15.11.6.4.3 ***NativeError*.prototype.message**

The initial value of the `message` property of the prototype for a given *NativeError* constructor is the empty String.

15.11.6.5 **Properties of *NativeError* Instances**

NativeError instances are ordinary objects that inherit properties from their *NativeError* prototype object and have a [[ErrorData]] internal data property whose initial value is **undefined**. The only specified use of [[ErrorData]] is to flag whether or not an Error or *NativeError* instance has been initialised by its constructor.

15.12 The JSON Object

The **JSON** object is a single ordinary object that contains two functions, **parse** and **stringify**, that are used to parse and construct JSON texts. The JSON Data Interchange Format is described in RFC 4627 <http://www.ietf.org/rfc/rfc4627.txt>. The JSON interchange format used in this specification is exactly that described by RFC 4627 with two exceptions:

- The top level *JSONText* production of the ECMAScript JSON grammar may consist of any *JSONValue* rather than being restricted to being a *JSONObject* or a *JSONArray* as specified by RFC 4627.
- Conforming implementations of **JSON.parse** and **JSON.stringify** must support the exact interchange format described in this specification without any deletions or extensions to the format. This differs from RFC 4627 which permits a JSON parser to accept non-JSON forms and extensions.

The value of the [[Prototype]] internal data property of the JSON object is the standard built-in Object prototype object (15.2.4). The JSON object has a [[JSONTag]] internal data property whose value is **true**. The value of the [[Extensible]] internal data property 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.

15.12.1 The JSON Grammar

JSON.stringify produces a String that conforms to the following JSON grammar. JSON.parse accepts a String that conforms to the JSON grammar.

15.12.1.1 **The JSON Lexical Grammar**

JSON is similar to ECMAScript source text in that it consists of a sequence of Unicode characters conforming to the rules of *SourceCharacter*. The JSON Lexical Grammar defines the tokens that make up a JSON text similar to the manner that the ECMAScript lexical grammar defines the tokens of an ECMAScript source text. The JSON Lexical grammar only recognises the white space character specified by the production *JSONWhiteSpace*. The JSON lexical grammar shares some productions with the ECMAScript lexical grammar. All nonterminal symbols of the grammar that do not begin with the characters “JSON” are defined by productions of the ECMAScript lexical grammar.

Deleted: NOTE The prototypes for the NativeError constructors do not themselves provide a `toString` function, but instances of errors will inherit it from the Error prototype object.¶

Deleted: n

Deleted: IsError

Deleted: NativeBrand

Deleted: BuiltinBrand

Deleted: value is

Deleted: NativeError

Deleted: z

Deleted: BuiltinError their [[Class]] internal property value is `"Error"`. *NativeError* instances have no spe

Deleted: value of the [[Class]] internal property of the

Deleted: n

Deleted: IsJSON

Deleted: NativeBrand

Deleted: BuiltinBrand

Deleted: **NativeJSON**

Deleted: **BuiltinJSON**

Deleted: is `"JSON"`

Deleted: property

Deleted: property

Syntax

JSONWhiteSpace **::**
<TAB>
<CR>
<LF>
<SP>

JSONString **::**
`"` *JSONStringCharacters*opt `"`

JSONStringCharacters **::**
*JSONStringCharacter JSONStringCharacters*opt

JSONStringCharacter **::**
SourceCharacter **but not one of** `"` **or** `\` **or** *U+0000* **through** *U+001F*
`\` *JSONEscapeSequence*

JSONEscapeSequence **::**
JSONEscapeCharacter
`u` *HexDigit HexDigit HexDigit HexDigit*

Deleted: *UnicodeEscapeSequence*

JSONEscapeCharacter **:: one of**
`" / \ b f n r t`

JSONNumber **::**
`-`opt *DecimalIntegerLiteral JSONFraction*opt *ExponentPart*opt

JSONFraction **::**
`.` *DecimalDigits*

JSONNullLiteral **::**
NullLiteral

JSONBooleanLiteral **::**
BooleanLiteral

15.12.1.2 The JSON Syntactic Grammar

The JSON Syntactic Grammar defines a valid JSON text in terms of tokens defined by the JSON lexical grammar. The goal symbol of the grammar is *JSONText*.

Syntax

JSONText **:**
JSONValue

JSONValue **:**
JSONNullLiteral
JSONBooleanLiteral
JSONObject
JSONArray
JSONString
JSONNumber

JSONObject **:**
`{ }`
`{` *JSONMemberList* `}`

JSONMember **:**
JSONString `:` *JSONValue*

JSONMemberList **:**
    JSONMember
    JSONMemberList **,** *JSONMember*

JSONArray **:**
    [ ]
    [*JSONElementList* **]**

JSONElementList **:**
    JSONValue
    JSONElementList **,** *JSONValue*

15.12.2 JSON.parse (text [, reviver])

The **parse** function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and **null**. JSON uses a more limited set of white space characters than *WhiteSpace* and allows Unicode code points U+2028 and U+2029 to directly appear in *JSONString* literals without using an escape sequence. The process of parsing is similar to 11.1.4 and 11.1.5 as constrained by the JSON grammar.

The optional *reviver* parameter is a function that takes two parameters, (*key* and *value*). It can filter and transform the results. It is called with each of the *key*/*value* pairs produced by the parse, and its return value is used instead of the original value. If it returns what it received, the structure is not modified. If it returns **undefined** then the property is deleted from the result.

1. Let *JText* be ToString(*text*).
2. ReturnIfAbrupt(*text*).
3. Parse *JText* interpreted as UTF-16 encoded Unicode characters using the grammars in 15.12.1. Throw a **SyntaxError** exception if *JText* did not conform to the JSON grammar for the goal symbol *JSONText*.
4. Let *scriptText* be the result of concatenating **"("**, *JText*, and **");"**.
5. Let *completion* be the result of parsing and evaluating *scriptText* as if it was the source text of an ECMAScript *Script* but using *JSONString* in place of *StringLiteral*. Note that since *JText* conforms to the JSON grammar this result will be either a primitive value or an object that is defined by either an *ArrayLiteral* or an *ObjectLiteral*.
6. Let *unfiltered* be *completion*.[[value]].
7. If IsCallable(*reviver*) is **true**, then
    a. Let *root* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
    b. Call CreateOwnDataProperty(*root*, the empty String, *unfiltered*).
    c. Return the result of calling the abstract operation Walk, passing *root* and the empty String. The abstract operation Walk is described below.
8. Else
    a. Return *unfiltered*.

Runtime Semantics: Walk Abstract Operation

The abstract operation Walk is a recursive abstract operation that takes two parameters: a *holder* object and the String *name* of a property in that object. Walk uses the value of *reviver* that was originally passed to the above parse function.

1. Let *val* be the result of Get(*holder*, *name*).
2. ReturnIfAbrupt(*val*).
3. If *val* is an object, then
    a. If *val* is an exotic Array object then
        i. Set *I* to 0.
        ii. Let *len* be the result of Get(*val*, **"length"**).
        iii. Assert: *len* is not an abrupt completion and its value is a positive integer.
        iv. Repeat while *I* < *len*,

Deleted: er
Deleted:
Deleted: ,
Comment [AWB15191]: TODO: What about the sta… semantic rules? Do they all apply to JSON text? Don't want to recognised __proto__, etc. It may be time to provide independent evaluation semantics for *JSONTe…*
Deleted: *unfiltered*
Deleted: *JText*
Deleted: *Program*
Deleted: abstraction operation
Deleted: (15.2)
Deleted: a new object created as if by the expression **new Object()**, where **Object** is the standard built-… constructor with that name
Deleted: the [[DefineOwnProperty]] internal method …
Deleted: with
Deleted: the PropertyDescriptor {[[Value]]:
Deleted: , [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false** as arguments
Deleted: calling the [[
Deleted:]] internal method of
Deleted:
Deleted: with argument
Comment [AWB12192]: TODO make this a more extensible condition to check
Deleted: the [[Class]] internal property of
Deleted: is **"Array"**
Deleted: has a [[IsArrayNativeBrandBuiltinBrand]] internal data property with value NativeArrayBuiltinArr…
Deleted: calling the [[Get]] internal method of *val* with argument **"length"**

1. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and ToString(*I*).
2. If *newElement* is **undefined**, then
   a. Let *status* be the result of calling the [[Delete]] internal method of *val* with ToString(*I*) as the argument.
3. Else
   a. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *val* with arguments ToString(*I*) and Property Descriptor {[[Value]]: *newElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
   b. NOTE The above is intentionally not a strict define and does not throw if status is **false**.
4. ReturnIfAbrupt(*status*).
5. Add 1 to *I*.

b. Else

   i. Let *keys* be an internal List of String values consisting of the names of all the own properties of *val* whose [[Enumerable]] attribute is **true**. The ordering of the Strings is the same as that used by the **Object.keys** standard built-in function.
   ii. For each String *P* in *keys* do,
      1. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and *P*.
      2. If *newElement* is **undefined**, then
         a. Let *status* be the result of calling the [[Delete]] internal method of *val* with *P* as the argument.
      3. Else
         a. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *val* with arguments *P* and Property Descriptor {[[Value]]: *newElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
         b. NOTE The above is intentionally not a strict define and does not throw if status is **false**.
      4. ReturnIfAbrupt(*status*).

4. Return the result of calling the [[Call]] internal method of *reviver* passing *holder* as *thisArgument* and with a List containing *name* and *val* as *argumentsList*.

It is not permitted for a conforming implementation of **`JSON.parse`** to extend the JSON grammars. If an implementation wishes to support a modified or extended JSON interchange format it must do so by defining a different parse function.

NOTE In the case where there are duplicate name Strings within an object, lexically preceding values for the same key shall be overwritten.

15.12.3 JSON.stringify (value [, replacer [, space]])

The **`stringify`** function returns a String in UTF-16 encoded JSON format representing an ECMAScript value. It can take three parameters. The *value* parameter is an ECMAScript value, which is usually an object or array, although it can also be a String, Boolean, Number or **null**. The optional *replacer* parameter is either a function that alters the way objects and arrays are stringified, or an array of Strings and Numbers that acts as a white list for selecting the object properties that will be stringified. The optional *space* parameter is a String or Number that allows the result to have white space injected into it to improve human readability.

These are the steps in stringifying an object:

1. Let *stack* be an empty List.
2. Let *indent* be the empty String.
3. Let *PropertyList* and *ReplacerFunction* be **undefined**.
4. If Type(*replacer*) is Object, then
   a. If IsCallable(*replacer*) is **true**, then
      i. Let *ReplacerFunction* be *replacer*.
   b. Else if *replacer* is an exotic Array object, then

Deleted: Call
Deleted: and **false**
Deleted: s
Deleted: Call
Deleted: , the
Deleted: , and **false**
Deleted: should
Deleted: be
Deleted: Call
Deleted: and **false**
Deleted: arguments
Deleted: Call
Deleted: , the
Deleted: , and **false**
Deleted: the **this** value
Deleted: n
Deleted: argument l
Deleted: consisting
Deleted: of
Deleted: The first parameter is required.
Deleted: the [[Class]] internal property of
Deleted: has a [[IsArrayNativeBrandBuiltinBrand]] internal data property with value NativeArrayBuiltinArray
Deleted: is **"Array"**

i. Let *PropertyList* be an empty internal List

ii. For each value *v* of a property of *replacer* that has an array index property name. The properties are enumerated in the ascending array index order of their names.

1. Let *item* be **undefined**.
2. If Type(*v*) is String then let *item* be *v*.
3. Else if Type(*v*) is Number then let *item* be ToString(*v*).
4. Else if Type(*v*) is Object then,
   a If *v* has a [[StringData]] or [[NumberData]] internal data property, then let *item* be ToString(*v*).
5. If *item* is not **undefined** and *item* is not currently an element of *PropertyList* then,
   a Append *item* to the end of *PropertyList*.

5. If Type(*space*) is Object then,
   a. If *space* has a [[NumberData]] internal data property, then,
      i. Let *space* be ToNumber(*space*).
   b. Else if *space* has a [[StringData]] internal data property, then,
      i. Let *space* be ToString(*space*).
6. If Type(*space*) is Number
   a. Let *space* be min(10, ToInteger(*space*)).
   b. Set *gap* to a String containing *space* occurrences of code unit 0x0020 (the Unicode space character). This will be the empty String if *space* is less than 1.
7. Else if Type(*space*) is String
   a. If the number of elements in *space* is 10 or less, set *gap* to *space* otherwise set *gap* to a String consisting of the first 10 elements of *space*.
8. Else
   a. Set *gap* to the empty String.
9. Let *wrapper* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
10. Call CreateOwnDataProperty(*wrapper*, the empty String, *value*).
11. Return the result of calling the abstract operation Str with the empty String and *wrapper*.

Runtime Semantics: Str Abstract Operation

The abstract operation *Str*(*key*, *holder*) has access to *ReplacerFunction* from the invocation of the `stringify` method. Its algorithm is as follows:

1. Let *value* be the result of Get(*holder*, *key*).
2. ReturnIfAbrupt(*value*).
3. If Type(*value*) is Object, then
   a. Let *toJSON* be the result of Get(*value*, `"toJSON"`).
   b. If IsCallable(*toJSON*) is **true**
      i. Let *value* be the result of calling the [[Call]] internal method of *toJSON* passing *value* as *thisArgument* and a List containing *key* as *argumentsList*.
      ii. ReturnIfAbrupt(*value*).
4. If *ReplacerFunction* is not **undefined**, then
   a. Let *value* be the result of calling the [[Call]] internal method of *ReplacerFunction* passing *holder* as the **this** value and with an argument list consisting of *key* and *value*.
   b. ReturnIfAbrupt(*value*).
5. If Type(*value*) is Object then,
   a. If *value* has an [[NumberData]] internal data property, then,
      i. Let *value* be ToNumber(*value*).
   b. Else if *value* has an [[StringData]] internal data property, then,
      i. Let *value* be ToString(*value*).
   c. Else if *value* has an [[BooleanData]] internal data property, then,
      i. Let *value* be the value of the [[BooleanData]] internal data property of *value*.
6. If *value* is **null** then return `"null"`.
7. If *value* is **true** then return `"true"`.
8. If *value* is **false** then return `"false"`.
9. If Type(*value*) is String, then return the result of calling the abstract operation Quote with argument *value*.
10. If Type(*value*) is Number

Deleted: the [[Class]] internal property of ...
Deleted: either …n
Deleted: IsString
Deleted: NativeBrand
Deleted: or an [[IsNumber]]
Deleted: whose value is either StringWrapper NumberWrapper
Deleted: the [[Class]] internal property of
Deleted: IsNumber
Deleted: NativeBrand
Deleted: is `"Number"`
Deleted: the [[Class]] internal property of
Deleted: IsString
Deleted: NativeBrand
Deleted: is `"String"`
Deleted: occurances
Deleted: characters
Deleted: characters …lements in *space* is 10 or less, s
Deleted: be …he result of the abstraction operation
Deleted: (15.2)
Deleted: a new object created as if by the expression `new Object()`, where `Object` is the standard built-in constructor with that name
Deleted: Call the [[DefineOwnProperty]] internal metho of …*rapper* with arguments the … the empty String, the
Deleted: calling the [[…et]] internal method of …*holder*
Deleted: calling the [[…et]] internal method of… …*a*
Deleted: the **this** value …nd with an argument … l
Deleted: the [[Class]] internal property of
Deleted: IsNumber
Deleted: NativeBrand
Deleted: is `"Number"`
Deleted: the [[Class]] internal property of
Deleted: NativeBrand
Deleted: IsString
Deleted: is `"String"`
Deleted: the [[Class]] internal property of
Deleted: NativeBrand
Deleted: IsBoolean
Deleted: is `"Boolean"`
Deleted: PrimitiveValue

a. If *value* is finite then return ToString(*value*).
b. Else, return **"null"**.

11. If Type(*value*) is Object, and IsCallable(*value*) is **false**
    a. If *value* is an exotic Array object, then
        i. Return the result of calling the abstract operation *JA* with argument *value*.
    b. Else, return the result of calling the abstract operation *JO* with argument *value*.
12. Return **undefined**.

Deleted: the [[Class]] internal property of

Deleted: has an [[IsArrayNativeBrandBuiltinBrand]] internal data property with value NativeArrayBuiltinArray

Deleted: is **"Array"**

Runtime Semantics: Quote Abstract Operation

The abstract operation Quote(*value*) wraps a String value in double quotes and escapes characters within it.

1. Let *product* be code unit 0x0022 (the Unicode double quote character).
2. For each code unit *C* in *value*
    a. If *C* is 0x0022 or 0x005C (the Unicode reverse solidus character)
        i. Let *product* be the concatenation of *product* and code unit 0x005C.
        ii. Let *product* be the concatenation of *product* and *C*.
    b. Else if *C* is backspace, formfeed, newline, carriage return, or tab
        i. Let *product* be the concatenation of *product* and code unit 0x005C (the Unicode backslash character).
        ii. Let *abbrev* be the string value corresponding to the value of *C* as follows:

backspace	**"b"**
formfeed	**"f"**
newline	**"n"**
carriage return	**"r"**
tab	**"t"**

        iii. Let *product* be the concatenation of *product* and *abbrev*.
    c. Else if *C* has a code unit value less than 0x0020 (the Unicode space character)
        i. Let *product* be the concatenation of *product* and code unit 0x005C (the Unicode backslash character).
        ii. Let *product* be the concatenation of *product* and **"u"**.
        iii. Let *hex* be the string result of converting the numeric code unit value of *C* to a String of four hexadecimal digits. Alphabetic hexadecimal digits are presented as lowercase characters.
        iv. Let *product* be the concatenation of *product* and *hex*.
    d. Else
        i. Let *product* be the concatenation of *product* and *C*.
3. Let *product* be the concatenation of *product* and code unit 0x0022 (the Unicode double quote character).
4. Return *product*.

Deleted: character

Deleted: the double quote character

Deleted: or

Deleted: backslash

Deleted: (the Unicode backslash charac

Deleted: *product* and the backslash character.

Deleted: and

Deleted: character

Deleted: vlaue

Deleted: is

Deleted: control character having a

Deleted: and

Deleted: and the double quote character

Runtime Semantics: JO Abstract Operation

The abstract operation *JO*(*value*) serializes an object. It has access to the *stack*, *indent*, *gap*, and *PropertyList* of the invocation of the stringify method.

1. If *stack* contains *value* then throw a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.
3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. If *PropertyList* is not **undefined**, then
    a. Let *K* be *PropertyList*.
6. Else
    a. Let *K* be an internal List of Strings consisting of the keys of all the own properties of *value* whose [[Enumerable]] attribute is **true and whose property key is a String value**. The ordering of the Strings is the same as that used by the **Object.keys** standard built-in function.
7. Let *partial* be an empty List.
8. For each element *P* of *K*.
    a. Let *strP* be the result of calling the abstract operation Str with arguments *P* and *value*.
    b. ReturnIfAbrupt(*strP*).
    c. If *strP* is not **undefined**

Deleted: ,

Deleted: *ReplacerFunction*, and *space*

Deleted: names

Deleted: should be

i. Let *member* be the result of calling the abstract operation Quote with argument *P*.
ii. Let *member* be the concatenation of *member* and the string `":"`.
iii. If *gap* is not the empty String
1. Let *member* be the concatenation of *member* and code unit 0x0020 (the Unicode space character).
iv. Let *member* be the concatenation of *member* and *strP*.
v. Append *member* to *partial*.

Deleted: colon character

9. If *partial* is empty, then
   a. Let *final* be `"{}"`.
10. Else
   a. If *gap* is the empty String
      i. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with code unit 0x002C (the Unicode comma character). A comma is not inserted either before the first String or after the last String.
      ii. Let *final* be the result of concatenating `"{"`, *properties*, and `"}"`.
   b. Else *gap* is not the empty String
      i. Let *separator* be the result of concatenating code unit 0x002C (the comma character), code unit 0x000A (the line feed character), and *indent*.
      ii. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
      iii. Let *final* be the result of concatenating `"{"`, code unit 0x000A (the line feed character), *indent*, *properties*, code unit 0x000A, *stepback*, and `"}"`.
11. Remove the last element of *stack*.
12. Let *indent* be *stepback*.
13. Return *final*.

Deleted: the line feed character

Runtime Semantics: JA Abstract Operation

The abstract operation *JA*(*value*) serializes an array. It has access to the *stack*, *indent*, and *gap* of the invocation of the stringify method. The representation of arrays includes only the elements between zero and `array.length` – 1 inclusive. Properties whose keys are not array indexes are excluded from the stringification. An array is stringified as an open left bracket, elements separated by comma, and a closing right bracket.

Deleted: , and *space*

Deleted: Named p

Deleted: are

1. If *stack* contains *value* then throw a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.
3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. Let *partial* be an empty List.
6. Assert: *value* is a standard array object and hence its `"length"` property is a non-negative integer.
7. Let *len* be the result of Get(*value*, `"length"`).
8. Let *index* be 0.
9. Repeat while *index* < *len*
   a. Let *strP* be the result of calling the abstract operation Str with arguments ToString(*index*) and *value*.
   b. ReturnIfAbrupt(*strP*).
   c. If *strP* is **undefined**
      i. Append `"null"` to *partial*.
   d. Else
      i. Append *strP* to *partial*.
   e. Increment *index* by 1.
10. If *partial* is empty ,then
   a. Let *final* be `"[]"`.
11. Else
   a. If *gap* is the empty String
      i. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with code unit 0x002C (the comma character). A comma is not inserted either before the first String or after the last String.
      ii. Let *final* be the result of concatenating `"["`, *properties*, and `"]"`.

Deleted: nativ

Deleted: e

Deleted: of calling the [[Get]] internal method of *value* w argument `"length"`.

b. Else
   i. Let *separator* be the result of concatenating code unit 0x002C (the comma character), code unit 0x000A (the line feed character), and *indent*.
   ii. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
   iii. Let *final* be the result of concatenating **"["**, code unit 0x000A (the line feed character), *indent*, *properties*, code unit 0x000A, *stepback*, and **"]"**.

Deleted: the line feed character

12. Remove the last element of *stack*.
13. Let *indent* be *stepback*.
14. Return *final*.

NOTE 1 JSON structures are allowed to be nested to any depth, but they must be acyclic. If *value* is or contains a cyclic structure, then the stringify function must throw a **TypeError** exception. This is an example of a value that cannot be stringified:

```
a = [];
a[0] = a;
my_text = JSON.stringify(a); // This must throw an 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`.

Deleted: `false`

NOTE 3 String values are wrapped in double quotes. The characters **"** and **** are escaped with **** prefixes. Control characters are replaced with escape sequences **\u**HHHH, or with the shorter forms, **\b** (backspace), **\f** (formfeed), **\n** (newline), **\r** (carriage return), **\t** (tab).

NOTE 4 Finite numbers are stringified as if by calling ToString(*number*). **NaN** and Infinity regardless of sign are represented as the String **null**.

NOTE 5 Values that do not have a JSON representation (such as **undefined** and functions) do not produce a String. Instead they produce the **undefined** value. In arrays these values are represented as the String **null**. In objects an unrepresentable value causes the property to be excluded from stringification.

NOTE 6 An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right brace. A property is a quoted String representing the key or property name, a colon, and then the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.

15.13 Binary Data Objects

15.13.1 The BinaryData Module

15.13.2 The BinaryData.Type Object

15.13.2.1 BinaryData.*ScalarType* Type Instance Objects

15.13.3 The BinaryData.ArrayType Object

15.13.4 The BinaryData.StructType Object

Comment [AWB10193]: This is a place holder for material in the Binary Data proposa http://wiki.ecmascript.org/doku.php?id=harmo :binary_data

15.13.5 ArrayBuffer Objects

15.13.5.1 Abstract Operations For ArrayBuffer Objects

15.13.5.1.1 AllocateArrayBuffer(constructor)

The abstract operation AllocateArrayBuffer with argument *constructor* is used to create an unitialised ArrayBuffer object. It performs the following steps:

1. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*constructor*, `"%ArrayBufferPrototype%"`, ([[ArrayBufferData]], [[ArrayBufferByteLength]])).
2. ReturnIfAbrupt(*obj*).
3. Set the [[ArrayBufferByteLength]] internal data property of *obj* to 0.
4. Return *obj*.

15.13.5.1.2 SetArrayBufferData(arrayBuffer, bytes)

The abstract operation SetArrayBufferData with arguments *arrayBuffer* and *bytes* is used to initialise the storage block encapsulated by an ArrayBuffer object. It performs the following steps:

1. Assert: *arrayBuffer* has an [[ArrayBufferData]] internal data property.
2. Assert: *bytes* is positive integer.
3. Let *block* be the result of CreateByteArrayBlock(*bytes*).
4. ReturnIfAbrupt(*block*).
5. Set *arrayBuffer's* [[ArrayBufferData]] to *block*.
6. Set *arrayBuffer*'s [[ArrayBufferByteLength]] internal data property to *bytes*.
7. Return *arrayBuffer*.

15.13.5.1.3 GetValueFromBuffer (arrayBuffer, byteIndex, type, isLittleEndian)

The abstract operation GetValueFromBuffer takes four parameters, an ArrayBuffer *arrayBuffer*, an integer *byteIndex*, a String *type*, and optionally a Boolean *isLittleEndian*. If *isLittleEndian* is not present, its default value is **undefined**. This operation performs the follow steps:

1. Assert: There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *valueType*.
2. Assert: *byteIndex* is a positive integer.
3. Let *block* be *arrayBuffer's* [[ArrayBufferData]] internal data property.
4. If *block* is **undefined** or **null**, then throw a **TypeError** exception.
5. Let *elementSize* be the Number value of the Element Size value specified in Table 36 for *valueType*.
6. Let *rawValue* be the *elementSize* bytes starting at *byteIndex* of *block*.
7. If *isLittleEndian* is **undefined**, set *isLittleEndian* to either **true** or **false**. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the SetValueInBuffer abstraction operation.
8. If *isLittleEndian* is **false**, reverse the order of the bytes of *rawValue*.
9. If *type* is "Float32" , then
   a. *rawValue* is interpreted as a little-endian bit string encoding of an IEEE 754-208 binary32 value.
   b. If *rawValue* is an IEEE 754-208 binary32 NaN value, return the **NaN** Number value.
   c. Return the Number value that is encoded by *rawValue*.
10. If *type* is "Float64" , then
    a. *rawValue* is interpreted as a little-endian bit string encoding of an IEEE 754-208 binary64 value.
    b. If *rawValue* is an IEEE 754-208 binary64 NaN value, return the **NaN** Number value.
    c. Return the Number value that is encoded by *rawValue*.
11. If the first characger of *type* is `"U"`, then
    a. Let *intValue* be the positive integer that is the result of interpreting *rawValue* as an unsigned little-endian binary number.
12. Else
    a. Let *intValue* be the signed integer that is the result of interpreting *rawValue* as a little-endian binary 2's complement number of bit length *elementSize* × 8.
13. Return *intValue*.

Deleted: ize

Deleted: <#>Assert: IsConstructor(*constructor*) is **true**.

Deleted:

Deleted: <#>Add a [[ArrayBufferData]] internal data property to *obj*.¶
Add

Deleted: a

Deleted: to

Deleted: <#>Set *obj*'s [[ArrayBufferData]] internal data property to **undefined**.¶
<#>Set *obj*'s [[ArrayBufferByteLength]] internal data property to 0.¶

Deleted: ize

Deleted: an Number value in the Uint32 range

Comment [AWB13194]: TODO: need to define abstract operations for allocting data blocks. Should throw a RangeError if alloc fails.

Deleted: *Length*

Deleted: Big

Deleted: internal

Deleted: *valueT*

Deleted: *Big*

Deleted: *Big*

Deleted: **false**

Deleted: *Big*

Deleted: *Big*

Deleted: *Big*

Deleted: **true**

Deleted: , taking into accont the value of *isBigEndian*

Deleted: any

Deleted: as

Deleted: , taking into accont the value of *isBigEndian*

Deleted: any

Deleted: as

Deleted: , taking into accont the value of *isBigEndian* *rawValue* as an unsigned binary number

Deleted: , taking into accont the value of *isBigEndian* *rawValue* as a binary 2's complement number of bit len *elementSize* × 4

15.13.5.1.4 SetValueInBuffer (arrayBuffer, byteIndex, type, value, isLittleEndian)

The abstract operation SetValueFromBuffer takes five parameters, an ArrayBuffer *arrayBuffer*, an integer *byteIndex*, a String *type*, a Number *value*, and optionally a Boolean *isLittleEndian*. If *isLittleEndian* is not present, its default value is **undefined**. This operation performs the follow steps:

1. Assert: There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *valueType*.
2. Assert: *byteIndex* is a positive integer.
3. Let *block* be *arrayBuffer's* [[ArrayBufferData]] internal data property.
4. If *block* is **undefined** or **null**, then throw a **TypeError** exception.
5. Let *elementSize* be the Number value of the Element Size value specified in Table 36 on the row with Element Type is *type*.
6. Let *rawValue* be the *elementSize* bytes starting at *byteIndex* of *arrayBuffer*.
7. If *isLittleEndian* is **undefined**, set *isLittleEndian* to either **true** or **false**. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the GetValueFromBuffer abstraction operation.
8. If *type* is "Float32" , then
   a. Set *rawValue* to the 4 bytes that are the result of converting *value* to IEEE-868-2005 binary32 format using "Round to nearest, ties to even" rounding mode. If *isLittleEndian* is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If *value* is **NaN**, *rawValue* is may be set of any implementation choosen non-signaling NaN encoding.
9. Else, if *type* is "Float64" , then
   a. Set *rawValue* to the 8 bytes that are the IEEE-868-2005 binary64 format encoding of *value*. If *isLittleEndian* is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If *value* is **NaN**, *rawValue* is may be set to any implementation choosen non-signaling NaN encoding.
10. Else,
    a. Let *n* be the Size Element value in Table 36 for the row containing the value of *type* as its Element Type entry.
    b. Let *convOp* be the abstract operation named in the Conversion Operation column in Table 36 for the row containing the value of *type* as its Element Type entry.
    c. Let *intValue* be the result of calling *convOp* with *value* as its argument .
    d. If *intValue* ≥ 0, then
       i. Let *rawBytes* be the *n*-byte binary encoding of *intValue*. If *isLittleEndian* is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order.
    e. Else,
       i. Let *rawBytes* be the *n*-byte binary 2's complement encoding of *intValue*. If *isLittleEndian* is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order.
11. Store the individual bytes of *rawBytes* in order starting at position *byteIndex* of *block*.
12. Return NormalCompletion (**undefined**).

15.13.5.2 The ArrayBuffer Constructor

When `ArrayBuffer` is called as a function rather than as a constructor, it creates and initialises a new ArrayBuffer object. Thus the function call `ArrayBuffer(…)` is equivalent to the object creation expression `new ArrayBuffer (…)` with the same arguments. However, if the **this** value value passed in the call is an Object with an [[ArrayBufferData]] internal data property whose value is **undefined**, it initialises the **this** value using the argument values. This permits `Array` to be used both as factory method and to perform constructor instance initialization.

The `ArrayBuffer` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified `ArrayBuffer` behaviour must include a `super` call to the `ArrayBuffer` constructor to initialise subclass instances.

15.13.5.2.1 ArrayBuffer(length)

ArrayBuffer called as function with argument *length* performs the following steps:

Deleted: NOTE The default number storag formal for all implementation is little endian, regardless of the native endianness of the underlying hardward.¶ 15.13.5.1.4
Deleted: Big
Deleted: internal
Deleted: *valueT*
Deleted: *Big*
Deleted: *Big*
Deleted: **false**
Deleted: *Big*
Deleted: *Big*
Deleted: <#>If *isBigEndian* is **true**, reverse t order of the bytes of *rawValue*.¶
Deleted: *Big*
Deleted: **true**
Deleted: *Big*
Deleted: **true**
Deleted: of
Deleted: ion
Deleted: o
Deleted: *Big*
Deleted: **true**
Deleted: *Big*
Deleted: **true**
Deleted: Normal
Deleted: Object Called as a Function
Deleted: it initializes its **this** value with the internal state that supports the `ArrayBuffer.prototype` internal methods This permits super invocation of the `ArrayBuffer` constructor by `ArrayBuffer` subclasses. If its this value is undefined,

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object or if *O* does not have an [[ArrayBufferData]] internal data property or if the value of *O's* [[ArrayBufferData]] internal data property is not **undefined**, then
   a. Throw a TypeError exception.
3. Assert: the value of *O's* [[ArrayBufferData]] internal data property is **undefined**.
4. Let *numberLength* be ToNumber(*length*).
5. Let *byteLength* be ToInteger(*numberLength*).
6. ReturnIfAbrupt(*byteLength*).
7. If *numberLength* ≠ *byteLength* or *byteLength* < 0, then throw a **RangeError** exception.
8. Return the result of SetArrayBufferData(*O*, *byteLength*).

15.13.5.2.2 new ArrayBuffer(... argumentsList)

ArrayBuffer called as part of a new expression it performs the following steps:

1. Let *F* be the ArrayBuffer function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If ArrayBuffer is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.13.5.3 Properties of the ArrayBuffer Constructor

The value of the [[Prototype]] internal data property of the ArrayBuffer constructor is the Function prototype object (15.3.3).

Besides its `length` property (whose value is 1), the ArrayBuffer constructor has the following properties:

15.13.5.3.1 ArrayBuffer.prototype

The initial value of ArrayBuffer.prototype is the ArrayBuffer prototype object (15.13.5.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.13.5.3.2 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 data property, then return **true**.
3. Return **false**.

15.13.5.3.3 ArrayBuffer[@@create] ( )

The @@create method of a ArrayBuffer function object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Return the result of calling AllocateArrayBuffer(*F*).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.13.5.4 Properties of the ArrayBuffer Prototype Object

The value of the [[Prototype]] internal data property of the ArrayBuffer prototype object is the standard built-in Object prototype object (15.2.4). The ArrayBuffer prototype object is an ordinary object. It does not have an [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal data property.

Deleted: <#>If Type(*O*) is neither Object or Undefined, then throw a **TypeError** exception¶
<#>If *O* is **undefined** or does not have an [[ArrayBufferData]] internal data property, then¶
Let *F* be this function object.

Deleted: <#>Let *argumentsList* be the be thebe the *argumentsList* argument of the [[Call]] internal method invoked *F*.¶
<#>Return the result of calling OrdinaryCreateFromConstructor (*F*, *argumentsList*). ¶
If

Deleted: not

Deleted: , then throw a **TypeError** exception,

Deleted: ToUint32

Deleted: Positive

Deleted: *length*

Deleted: <#>15.13.5.23 The ArrayBuffer Constructor¶
When ArrayBuffer is called as part of a new expression is a constructor: it initialises the newly created object.¶
The ArrayBuffer constructor is designed to be subclassable. It may be used as the value of an `exter` clause of a class declaration. Subclass constructors tha intended to inherit the specified ArrayBuffer behaviour must include a `super` call to ArrayBuffer.¶
15.13.5.23.1

Deleted: len

Deleted: args

Deleted: with argument *length*

Deleted: be the be the

Deleted: with arguments

Deleted:

Deleted: and

Deleted: Map

Deleted: a

Deleted: The [[Prototype]] internal data property of the newly constructed object is set to the original ArrayBuf prototype object, the one that is the initial value of ArrayBuffer.prototype (16.1.3.1). The [[Class]] internal data property of the newly constructed object is set to "ArrayBuffer". The [[Extensible]] internal data property the newly constructed object is set to **true**.¶
The length property of the newly constructed object is s to ToUInt32(len).¶

Deleted: 15.3.3

Deleted: `the internal properties and the`

Deleted: 15.13.5.4

Deleted: 16

Deleted: <#>If *arg* is an exotic Array object, then retur

Deleted: ,

Deleted: <#>If IsConstructor(*F*) is **false**, then throw a

Deleted: false

Deleted: <#>Properties of the ArrayBuffer

Deleted: The [[Class]] internal data property of the ne

Deleted: The [[Extensible]] internal data property of th

15.13.5.4.1 ArrayBuffer.prototype.constructor

The initial value of ArrayBuffer.prototype.constructor is the standard built-in ArrayBuffer constructor.

15.13.5.4.2 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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ArrayBufferData]] internal data property throw a **TypeError** exception.
4. If the value of *O*'s [[ArrayBufferData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *length* be the value of *O*'s [[ArrayBufferByteLength]] internal data property.
6. Return *length*.

15.13.5.4.3 ArrayBuffer.prototype.slice (start , end)

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ArrayBufferData]] internal data property throw a **TypeError** exception.
4. If the value of *O*'s [[ArrayBufferData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *len* be the value of *O*'s [[ArrayBufferByteLength]] internal data property.
6. Let *relativeStart* be ToInteger(*start*).
7. ReturnIfAbrupt(*relativeStart*).
8. If *relativeStart* is negative, let *first* be max((*len* + *relativeStart*),0); else let *first* be min(*relativeStart*, *len*).
9. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ToInteger(*end*).
10. ReturnIfAbrupt(*relativeEnd*).
11. If *relativeEnd* is negative, let *final* be max((*len* + *relativeEnd*),0); else let *final* be min(*relativeEnd*, *len*).
12. Let *newLen* be max(*final*-*first*,0).
13. Let *ctor* be the result of GetMethod(*O*, `"constructor"`).
14. ReturnIfAbrupt(*ctor*).
15. If IsConstructor(*ctor*) is **false**, then throw a **TypeError** exception.
16. Let *new* be the result of calling the [[Construct]] internal method of *ctor* with a new List containing the single element *newLen*.
17. ReturnIfAbrupt(*new*).
18. If *new* does not have a [[ArrayBufferData]] internal data property throw a **TypeError** exception.
19. If the value of *new*'s [[ArrayBufferData]] internal data property is **undefined**, then throw a **TypeError** exception.
20. Let *fromBuf* be the value of *O*'s [[ArrayBufferData]] internal data property.
21. Let *toBuf* be the value of *new*'s [[ArrayBufferData]] internal data property.
22. Let *status* be the result of CopyBlockElements(*fromBuf*, *first*, *toBuf*, 0, *newLen*).
23. ReturnIfAbrupt(*status*).
24. Return *new*.

15.13.5.4.4 ArrayBuffer.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value `"ArrayBuffer"`.

15.13.5.5 Properties of the ArrayBuffer Instances

ArrayBuffer instances inherit properties from the ArrayBuffer prototype object. ArrayBuffer instances each have a [[ArrayBufferData]] internal data property and a [[ArrayBufferByteLength]] internal data property.

Comment [AWB13195]: byteLength needs be an accessor both to comply with WebIDL requirements and to support the Kronos neutering strawman requirements.

Deleted: `size`

Deleted: ByteLength

Deleted: be

Deleted: be

Comment [AWB13196]: TODO: need to define abstract operations for alloctíng and manipulating data blocks.

Deleted: .

Deleted: and their [[Class]] internal data property value is "ArrayBuffer"

Deleted: After initialisation by the ArrayBuffe constructor,

Deleted: also

Deleted: ArrayBuffer instances also have th following properties.

15.13.6 *TypedArray* Objects

TypedArray objects present an array-like view of any underlying binary data buffer. Each element of a TypedArray instance has the same underlying binary scalar data type. There is a distinct *TypedArray* constructor for each of the nine support element types. For each constructor in the Table 36 is a separate *TypedArray* constructor object, with corresponding prototype object and instances.

Table 36 – The *TypedArray* Constructors

Constructor Name	Element Type	Element Size	Conversion Operation	Description	Equivalent C Type
Int8Array	Int8	1	ToInt8	8-bit 2's complement signed integer	signed char
Uint8Array	Uint8	1	ToUint8	8-bit unsigned integer	unsigned char
Uint8ClampedArray	Uint8C	1	ToUint8Clamp	8-bit unsigned integer (clamped)	unsigned char
Int16Array	Int16	2	ToInt16	16-bit 2's complement signed integer	Short
Uint16Array	Uint16	2	ToUint16	16-bit unsigned integer	unsigned short
Int32Array	Int32	4	ToInt32	32-bit 2's complement signed integer	Int
Uint32Array	Uint32	4	ToUint32	32-bit unsigned integer	unsigned int
Float32Array	Float32	4		32-bit IEEE floating point	Float
Float64Array	Float64	8		64-bit IEEE floating point	Double

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.

15.13.6.1 The %TypedArray% Intrinsic Object

The %TypedArray% intrinsic object is a constructor-like function object that all of the *TypedArray* constructor object inherit from. %TypedArray% and its corresponding prototype object provide common properties that are inherited by all *TypeArray* constructors and their instances. The %TypedArray% intrinsic does not have a global name or appear as a property of the global object.

However, if the **this** value value passed in the call is an Object with an [[ViewedArrayBuffer]] internal data property whose value is **undefined**, it initializes the **this** value using the argument values. This permits super invocation of the ***TypedArray*** constructors by ***TypedArray*** subclasses.

The %TypedArray% intrinsic function object is designed to act as the superclass of the various *TypedArray* constructors. Those constructors use %TypedArray% to initialize their instances by invoking %TypedArray% as if by making a **super** call. The %TypedArray% intrinsic function is not designed to be directly called in any other way. If %TypedArray% is directly called or called as part of a **new** expression an exception is thrown.

The actual behaviour of a **super** call of %TypedArray% depends upon the number and kind of arguments that are passed to it.

15.13.6.1.1 %TypedArray% (length)

This description applies if and only if the %TypedArray% function is called with exactly one argument and the Type of the argument is not Number.

%TypedArray% called with argument *length* performs the following steps:

Deleted: 15.13.5.5.1 byteLength¶ The byteLength property of this ArrayBuffer object is a data property whose value is the length of the ArrayBu in bytes, as fixed at construction time.¶ The length property has the attributes { [[Writable]]: fal [[Enumerable]]: **false**, [[Configurable]]: **false** }.¶

Deleted: <#>15.13.6 *TypedArray* Object Structures¶

Deleted:

Deleted: following table

Deleted: *TypeArray*

Deleted: Each of these *TypeArrayTypedArray* constructor objects has the structure described below, differing only in the name used as the constructor nam instead of *TypeArrayTypedArray*, in Table 35Table 35Table 36XXXXXXX.

Deleted: Element

Deleted: Size

Comment [AWB13197]: TODO

Comment [AWB13198]: TODO

Comment [AWB13199]: Issue: this is a made up na What should it be?

Comment [AWB13200]: TODO

Deleted: *TypeArray*

Deleted: <#>15.13.6.1 *TypeArrayTypedArray* Constructors Called as a Function¶

Deleted: When a *TypeArrayTypedArray* constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the constructor as a function is equivalent to calling it as a constructor with the same arguments.¶

Deleted: its **this** value

Deleted: with the internal state that supports the ***TypedArray*.prototype** internal methods

Deleted: If the this value is undefined,

Deleted: *TypedArray*

Deleted: as function

1. Assert: Type(*length*) is not Object.
2.
3. Let *O* be the **this** value.
4. If Type(*O*) is not Object or if *O* does not have an [[ViewedArrayBuffer]] internal data property or if the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then
   a. Throw a **TypeError** exception.
5. If *O* does not have an [[TypedArrayName]] internal data property, then throw a **TypeError** exception.
6. Let *constructorName* be the string value *O*'s [[TypedArrayName]] internal data property.
7. Let *elementType* be the string value of the Element Type value in Table 36 for *constructorName*.
8. Let *numberLength* be ToNumber(*length*).
9. Let *elementLength* be ToInteger(*numberLength*).
10. ReturnIfAbrupt(*elementLength*).
11. If *numberLength* ≠ *elementLength* or *elementLength* < 0, then throw a **RangeError** exception.
12. Let *data* be the result of calling AllocateArrayBuffer(%ArrayBuffer%).
13. ReturnIfAbrupt(*data*).
14. Let *elementSize* be the Size Element value in Table 36 for *constructorName*.
15. Let *byteLength* be *elementSize* × *elementLength*.
16. Let *status* be the result of SetArrayBufferData(*data*, *byteLength*)5.
17. ReturnIfAbrupt(*status*).
18. Set *O's* [[ViewedArrayBuffer]] to *data*.
19. Set *O*'s [[ByteLength]] internal data property to *byteLength*.
20. Set *O*'s [[ByteOffset]] internal data property to 0.
21. Set *O*'s [[ArrayLength]] internal data property to *elementLength*.
22. Return *O*.

15.13.6.1.2 %TypedArray% (typedArray)

This description applies if and only if the %TypedArray% function is called with exactly one argument and the Type of the argument is Object and the object has a [[ViewedArrayBuffer]] internal data property.

%TypedArray% called with argument *typedArray* performs the following steps:

1. Assert: Type(*typedArray*) is Object and *typedArray* has a [[ViewedArrayBuffer]] internal data property.
2. Let *srcArray* be *typedArray*.
3. Let *O* be the **this** value.
4. If Type(*O*) is not Object or if *O* does not have an [[ViewedArrayBuffer]] internal data property or if the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then
   a. Throw a **TypeError** exception.
5. If *O* does not have an [[TypedArrayName]] internal data property, then throw a **TypeError** exception.
6. Let *constructorName* be the string value *O*'s [[TypedArrayName]] internal data property.
7. Let *elementType* be the string value of the Element Type value in Table 36 for *constructorName*.
8. Let *elementLength* be the value of *srcArray's* [[ArrayLength]] internal data property.
9. Let *srcName* be the string value *srcArray's* [[TypedArrayName]] internal data property.
10. Let *srcType* be the string value of the Element Type value in Table 36 for *srcName*.
11. Let *srcData* be the value of *srcArray's* [[ViewedArrayBuffer]] internal data property.
12. Let *data* be the result of calling CloneArrayBuffer(*srcData*, *srcType*,*elementType*, *elementLength*).
13. ReturnIfAbrupt(*data*).
14. Let *elementSize* be the Size Element value in Table 36 *constructorName*.
15. Let *byteLength* be *elementSize* × *elementLength*.
16. Let *data* be the result of SetArrayBufferData(*data*, *byteLength*).
17. ReturnIfAbrupt(*data*).
18. If the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then throw a **TypeError** exception.
19. Set *O's* [[ViewedArrayBuffer]] to *data*.
20. Set *O*'s [[ByteLength]] internal data property to *byteLength*.
21. Set *O*'s [[ByteOffset]] internal data property to 0.
22. Set *O*'s [[ArrayLength]] internal data property to *elementLength*.
23. Return *O*.

Comment [AWB13201]: Issue: we need t decide whether we are going to fully apply WebIDL's complex set of overload resolution rules.

Deleted: ;

Moved down [6]: Let *elementType* be the st value of the Element Type value in Table 35Tab 35Table 36 for this *TypedArray* constructor.¶

Deleted: <#>If Type(*O*) is neither Object or Undefined, then throw a **TypeError** exception¶

Deleted: If *O* is **undefined** or does not have a [[TypedArrayData]][[ViewedArrayBuffer]] inte data property, then

Deleted: Let *F* be this function object.

Deleted: <#>Let *argumentsList* be the be thebe the *argumentsList* argument of the [[Ca internal method that invoked *F*.¶
<#>Return the result of calling OrdinaryConstruct (*F*, *argumentsList*). ¶

Moved (insertion) [6]

Deleted: Table 35Table 35Table 36

Deleted: this

Deleted: *TypedArray* constructor

Deleted: <#>If the value of *O's* [[TypedArrayData]][[ViewedArrayBuffer]] inte data property is not **undefined**, then throw a **TypeError** exception,¶

Deleted: Uint32

Deleted: Positive

Deleted: *length*

Deleted: Table 36Table 35Table 35Table 36

Deleted: this *TypedArray* constructor

Deleted: [[TypedArrayData]]

Deleted: <#>Set *O's* [[TypedArrayElementKind]] internal data prope

Deleted: ***TypedArray***

Deleted: *TypedArray*

Deleted: as function

Comment [AWB13202]: Issue: we need t

Deleted: ;

Deleted: [[TypedArrayData]]

Deleted: <#>Let *elementType* be the string

Deleted: <#>If Type(*O*) is neither Object or

Deleted: the value of *srcArray's*

Deleted: [[TypedArrayData]]

Comment [AWB13203]: TODO

Deleted: Table 36Table 35Table 35Table 36

Deleted: for this *TypedArray* constructor

Deleted: *w*

Deleted: [[TypedArrayData]]

Deleted: <#>Set *O*'s

15.13.6.1.3 %TypedArray% (array)

This description applies if and only if the %TypedArray% function is called with exactly one argument and the Type of the argument is Object and the object does not have either a [[ViewedArrayBuffer]] or a [[ArrayBufferData]] internal data property.

%TypedArray% called with argument *array* performs the following steps:

1. Assert: Type(*array*) is Object and *array* does not have either a [[ViewedArrayBuffer]] or a [[ArrayBufferData]] internal data property.
2. Let *O* be the **this** value.
3. Let *srcArray* be *array*.
4. If Type(*O*) is not Object or if *O* does not have an [[ViewedArrayBuffer]] internal data property or if the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then
   a. Throw a **TypeError** exception.
5. If *O* does not have an [[TypedArrayName]] internal data property, then throw a **TypeError** exception.
6. Let *constructorName* be the string value *O*'s [[TypedArrayName]] internal data property.
7. Let *elementType* be the string value of the Element Type value in Table 36 for *constructorName*.
8. Let *arrayLength* be the result of Get(*array*, `"length"`).
9. Let *numberLength* be ToNumber(*arrayLength*).
10. Let *elementLength* be ToInteger(*numberLength*).
11. ReturnIfAbrupt(*elementLength*).
12. If *numberLength* ≠ *elementLength* or *elementLength* < 0, then throw a **TypeError** exception.
13. Let *data* be the result of calling AllocateArrayBuffer(%ArrayBuffer%).
14. ReturnIfAbrupt(*data*).
15. Let *elementSize* be the Size Element value in Table 36 for *constructorName*.
16. Let *byteLength* be *elementSize* × *elementLength*.
17. Let *status* be the result of SetArrayBufferData(*data*, *byteLength*)
18. Let *k* be 0.
19. Repeat, while *k* < *elementLength*
   a. Let *Pk* be ToString(*k*).
   b. Let *kValue* be the result of Get(*O*, *Pk*).
   c. Let *kNumber* be ToNumber(*kValue*).
   d. ReturnIfAbrupt(*kNumber*).
   e. Perform SetValueInBuffer(*data*, *k* × *elementSize*, *elementType*, *kNumber*).
   f. Increase *k* by 1.
20. If the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then throw a **TypeError** exception.
21. Set *O's* [[ViewedArrayBuffer]] to *data*.
22. Set *O*'s [[ByteLength]] internal data property to *byteLength*.
23. Set *O*'s [[ByteOffset]] internal data property to 0.
24. Set *O*'s [[ArrayLength]] internal data property to *elementLength*.
25. Return *O*.

15.13.6.1.4 %TypedArray% (buffer, byteOffset=0, length=undefined)

This description applies if and only if the %TypedArray% function is called with more than one argument or if it is called with exactly one argument and the Type of the argument is Object and the object has a [[ArrayBufferData]] internal data property.

%TypedArray% called with arguments *buffer*, *byteOffset*, and *length* performs the following steps:

1. Assert: Type(*buffer*) is Object and *buffer* has a [[ArrayBufferData]] internal data property.
2. Let *O* be the **this** value.
3. If Type(*O*) is not Object or if *O* does not have an [[ViewedArrayBuffer]] internal data property or if the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then
   a. Throw a **TypeError** exception.
4. If *O* does not have an [[TypedArrayName]] internal data property, then throw a **TypeError** exception.
5. Let *constructorName* be the string value *O*'s [[TypedArrayName]] internal data property.
6. Let *elementType* be the string value of the Element Type value in Table 36 for *constructorName*.

Comment [AWB13204]: Issue: The khronos spec. allows array-likes here. Should we also recognise iterables and use @@iterator for them? This algorithm current doesn't deal with iterables.

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: as function

Deleted: ;

Deleted: [[TypedArrayData]]

Comment [AWB13205]: Issue: we need to decide whether we are going to fully apply WebIDL's complex of overload resolution rules.

Deleted: <#>If Type(*O*) is neither Object or Undefined, then throw a **TypeError** exception¶

Deleted: <#>Let *elementType* be the string value of the Element Type value in Table 35Table 35Table 36 for this *TypedArray* constructor.¶
<#>If *O* is **undefined** or does not have an [[TypedArrayData]][[ViewedArrayBuffer]] internal data property, then¶
<#>Let *F* be this function object.¶
<#>Let *argumentsList* be the be thebe the *argumentsLi* argument of the [[Call]] internal method that invoked *F*.
<#>Return the result of calling OrdinaryConstruct (*F*, *argumentsList*). ¶
<#>If the value of *O's* [[TypedArrayData]][[ViewedArrayBuffer]] internal data property is not **undefined**, then throw a **TypeError** exception,¶

Deleted: Uint32

Deleted: Positive

Deleted: *arrayLength*

Deleted: this *TypedArray* constructor

Deleted: [[TypedArrayData]]

Deleted: <#>Set *O*'s [[TypedArrayElementKind]] inter data property to *elementType*.¶

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: as function

Deleted: *length*

Comment [AWB13206]: Issue: we need to decide whether we are going to fully apply WebIDL's complex of overload resolution rules.

Deleted: ;

Deleted: *typedArray*

Deleted: <#>If Type(*O*) is neither Object or Undefined, then throw a **TypeError** exception¶
<#>Let *elementType* be the String value of the Element T value in Table 35Table 35Table 36 for this *TypedArray* constructor.¶
<#>Let *elementSize* be the Number value of the Element value in Table 35Table 35Table 36 for this *TypedArray* constructor.¶

7. Let *elementSize* be the Number value of the Element Size value in Table 36 for *constructorName*.
8. Assert: the value of *O's* [[ViewedArrayBuffer]] internal data property is **undefined**.
9. Let *offset* be ToInteger(*byteOffset*).
10. ReturnIfAbrupt(*offset*).
11. If *offset* < 0, then throw a **RangeError** exception.
12. If *offset* modulo *elementSize* ≠ 0, then throw a **RangeError** exception.
13. Let *bufferByteLength* be the value of *buffer's* [[ArrayBufferByteLength]] internal data property.
14. If *offset* + *elementSize* ≥ *bufferByteLength*, then throw a **RangeError** exception.
15. If *length* is **undefined**, then
    a. If *bufferByteLength* modulo *elementSize* ≠ 0, then throw a **RangeError** exception.
    b. Let *newByteLength* be *bufferByteLength* – *offset*.
16. Else,
    a. Let *numberLength* be ToNumber(*length*).
    b. Let *newLength* be ToInteger (*numberLength*).
    c. ReturnIfAbrupt(*newLength*).
    d. If *numberLength* ≠ *newLength* or *newLength* < 0, then throw a **RangeError** exception.
    e. Let *newByteLength* be *newLength* × *elementSize*.
    f. If *offset*+*newByteLength* > *bufferByteLength*, then throw a **RangeError** exception.
17. If the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then throw a **TypeError** exception.
18. Set *O's* [[ViewedArrayBuffer]] to *buffer*.
19. Set *O*'s [[ByteLength]] internal data property to *newByteLength*.
20. Set *O*'s [[ByteOffset]] internal data property to *offset*.
21. Set *O*'s [[ArrayLength]] internal data property to *newByteLength* / *elementSize* .
22. Return *O*.

15.13.6.1.5 %TypedArray% (binary data stuff)

TODO: this is a place holder assuming that we may need to construct TypedArrays from binary data objects.

15.13.6.1.6 %TypedArray% (all other argument combinations)

If the %TypedArray% function is called with arguments that do match any of the preceeding argument descriptions a **TypeError** exception is thrown.

15.13.6.2 Properties of the %TypedArray% Intrinsic Object

the %TypedArray% intrinsic object is a built-in function object. The value of the [[Prototype]] internal data property of %TypedArray% is the Function prototype object (15.3.3).

Besides a `length` property (whose value is 3), %TypedArray% has the following properties:

15.13.6.2.1 %TypedArray%.prototype

The initial value of %TypedArray%.prototype is the %TypedArrayPrototype% intrinsic object (15.13.6.3).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.13.6.2.2 %TypedArray%.of (...items)

When the `of` method is called with any number of arguments, the following steps are taken:

12. Let *lenValue* be the result of Get(*items*, `"length"`).
13. Let *len* be ToInteger(*lenValue*).
14. Let *C* be the **this** value.
15. If IsConstructor(*C*) is **true**, then
    a. Let *newObj* be the result of OrdinaryConstruct(*C*, (*len*)).

Deleted: Uint32
Deleted: Positive
Comment [AWB13207]: Issue, the order c… exceptions that may be thrown during argum… validation is probably not currently the same … would be produced by WebIDL overloading p… Khronos validation rules (if that order is even well defined??). It isn't clear that this differen… is very important.
Deleted: Positive
Deleted: Uint32
Deleted: *length*
Deleted: [[TypedArrayData]]
Deleted: ,
Deleted: [[TypedArrayData]]
Deleted: <#>Set *O*'s [[TypedArrayElementKind]] internal data prope… to *elementType*.¶
Comment [AWB13208]: TODO: this is a place holder assuming that we may need to construct TypedArrays from binary data obje…
Deleted: ***TypedArray***
Deleted: <#>15.13.6.2 The ***TypeArrayTypedArray*** Constructors¶ When a *TypedArray* function is called as part… a new expression, it is a constructor: it initiali… the newly created object.¶ The *TypedArray* constructors is designed to b… subclassable. They may be used as the value… an **extends** clause of a class declaration. Subclass constructors that intended to inherit… the specified *TypedArray* behaviour must include a **super** call to the *TypedArray*.¶
Formatted: Font: (Default) Courier New
Deleted: When a *TypeArray* constructor is
Deleted: ***TypeArray***
Deleted: ***TypedArray*** **Constructors**
Deleted: each
Deleted: TypeArray
Deleted: constructor
Deleted: 15.3.4
Deleted: 15.3.3
Deleted: the internal properties and the
Deleted: each *TypeArrayTypedArray*
Deleted: ***TypeArray***
Deleted: ***TypedArray***
Deleted: *TypeArray*
Deleted: *TypedArray*
Deleted: corresponding *TypeArrayTypedArr…*
Deleted: 15.13.6.3
Comment [AWB7209]: It would be nice to…

b. Let *A* be ToObject(*newObj*).

16. Else,
    a. Throw a **TypeError** exception.
17. ReturnIfAbrupt(*A*).
18. Let *k* be 0.
19. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kValue* be the result of Get(*items*, *Pk*).
    c. Let *defineStatus* be the result of Put(*A*,*Pk*, *kValue*.[[value]], **true**).
    d. ReturnIfAbrupt(*defineStatus*).
    e. Increase *k* by 1.
20. Return *A*.

The **length** property of the **of** method is **0**.

NOTE 1 The *items* argument is assume to be a well-formed rest argument value.

NOTE 2 The **of** function is an intentionally generic factory method; it does not require that its **this** value be a *TypedArray* constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument. However, it does assume that constructor creates and initializes a length property that is initialized to its argument value.

15.13.6.2.3 %TypedArray%.from (source , mapfn=undefined, thisArg=undefined)

When the **from** method is called with argument *source*, and optional arguments *mapfn* and *thisArg*, the following steps are taken:

1. Let *C* be the **this** value.
2. If IsConstructor(*C*) is **false**, then throw a **TypeError** exception.
3. Let *items* be ToObject(*source*).
4. ReturnIfAbrupt(*items*).
5. If *mapfn* is **undefined**, then let *mapping* be **true**.
6. else
    a. If IsCallable(*mapfn*) is **false**, throw a **TypeError** exception.
    b. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
    c. Let *mapping* be **true**
7. Let *usingIterator* be the result of HasProperty(*items*, @@Iterator).
8. ReturnIfAbrupt(*usingIterator*).
9. If *usingIterator* is **true**, then
    a. Let *iterator* be the result of performing GetIterator(*obj*).
    b. ReturnIfAbrupt(*iterator*).
    c. Let *values* be a new empty List.
    d. Let *done* be **false**.
    e. Repeat, while *done* is **false**
        i. Let *next* be the result of IteratorNext(*iterator*).
        ii. ReturnIfAbrupt(*next*).
        iii. Let *done* be IteratorComplete(*next*).
        iv. ReturnIfAbrupt(*done*).
        v. If *done* is **false**, then
            1. Let *nextValue* be IteratorValue(*next*).
            2. ReturnIfAbrupt(*nextValue*).
            3. Append *nextValue* to the end of the List *values*.
    f. Let *len* be the number of elements in *values*.
    g. Let *newObj* be the result of OrdinaryConstruct(*C*, (*len*)).
    h. Let *A* be ToObject(*newObj*).
    i. ReturnIfAbrupt(*A*).
    j. Let *k* be 0.
    k. Repeat, while *k* < *len*
        i. Let *Pk* be ToString(*k*).
        ii. Let *kValue* be the first element of *values* and remove that element from *list*.

Comment [AWB7210]: It would be nice to have a m explicit way to create a collection with a pre-specified number of elements.

iii. If *mapping* is **true**, then
1. Let *mappedValue* be the result of calling the [[Call]] internal method of *mapfn* with *T* as *thisArgument* and a List containing *kValue* as *argumentsList*.
2. ReturnIfAbrupt(*mappedValue*).
iv. Else, let *mappedValue* be *kValue*.
v. Let *putStatus* be the result of Put(*A*, *Pk*, *mappedValue*, **true**).
vi. ReturnIfAbrupt(*putStatus*).
vii. Increase *k* by 1.
l. Asert: *values* is now an empty List.
m. Return *A*.
10. Assert: *items* is not an Iterator so assume it is Array-like.
11. Let *lenValue* be the result of Get(*items*, `"length"`).
12. Let *len* be ToInteger(*lenValue*).
13. ReturnIfAbrupt(*len*).
14. Let *newObj* be the result of OrdinaryConstruct(*C*, (*len*)).
15. Let *A* be ToObject(*newObj*).
16. ReturnIfAbrupt(*A*).
17. Let *k* be 0.
18. Repeat, while *k* < *len*
a. Let *Pk* be ToString(*k*).
b. Let *kValue* be the result of Get(*items*, *Pk*).
c. ReturnIfAbrupt(*kValue*).
d. If *mapping* is **true**, then
i. Let *mappedValue* be the result of calling the [[Call]] internal method of *mapfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *items* as *argumentsList*.
ii. ReturnIfAbrupt(*mappedValue*).
e. Else, let *mappedValue* be *kValue*.
f. Let *putStatus* be the result of Put(*A*, *Pk*, *mappedValue*, **true**).
g. ReturnIfAbrupt(*putStatus*).
h. Increase *k* by 1.
19. Return *A*.

NOTE The `from` function is an intentionally generic factory method; it does not require that its **this** value be the Array constructor. Therefore it can be transferred to or inherited by any other constructors that may be called with a single numeric argument. This function uses [[Put]] to store elements into a newly created object and assume that the constructor sets the `length` property of the new object to the argument value passed to it.

15.13.6.2.4 %TypedArray%[@@create] ( )

The @@create method of %TypedArray% performs the following steps:

1. Let *F* be the **this** value.
2. If Type(*F*) is not Object, then throw a **TypeError** exception.
3. If *F* does not have a [[TypedArrayConstructor]] internal data property, then throw a **TypeError** exception.
4. Let *proto* be the result of GetPrototypeFromConstructor(*F*, `"%TypedArrayPrototype%"`).
5. ReturnIfAbrupt(*proto*).
6. Let *obj* be the result of calling IntegerIndexedObjectCreate (*proto*).
7. Add a [[ViewedArrayBuffer]] internal data property to *obj* and set its initial value to **undefined**.
8. Add a [[TypedArrayName]] internal data property to *obj* and set its initial value to the value of *F's* [[TypedArrayConstructor]] internal data property.
9. Add a [[ByteLength]] internal data property to *obj* and set its initial value to 0.
10. Add a [[ByteOffset]] internal data property to *obj* and set its initial value to 0.
11. Add a [[ArrayLength]] internal data property to *obj* and set its initial value to 0.
12. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

Comment [AWB7211]: It would be nice to have a more explicit way to create a collectio with a pre-specified number of elements.

Deleted: *TypedArray*

Deleted: a *TypedArray* function object *F*

Deleted: ,

Deleted: <#>If IsConstructor(*F*) is **false**, the throw a **TypeError** exception.¶
<#>Let *proto* be the result of Get(*F*, `"prototype"`).¶

Deleted: <#>If Type(*proto*) is not Object, the
<#>If *constructor* has a [[Realm]] internal d property, let *realm* be *F's* [[Realm]].¶
<#>Else,¶
<#>Let *ctx* be the running execution cont
<#>Let *realm* be be *ctx's* Realm.¶
<#>Let *proto* be *realm's* intrinsic object nam `"%TypedArrayPrototype%"`.¶

Deleted: IndexedDelegatorCreate

Deleted: [[TypedArrayData]]

Deleted: ElementKind

Deleted:

Deleted: **undefined**

Deleted: <#>Set *obj*'s [[TypedArrayData]] internal data property to **undefined**.¶

Deleted: **false**

15.13.6.3 Properties of the %TypedArrayPrototype% Object

The value of the [[Prototype]] internal data property of the %TypedArrayPrototype% object is the standard built-in Object prototype object (15.2.4). The %TypedArrayPrototype% object is an ordinary object. It does not have a [[ViewedArrayBuffer]] or or any other of the internal data properties that are specific to *TypedArray* instance objects.

15.13.6.3.1 %TypedArray%.prototype.constructor

The initial value of %TypedArray%.prototype.constructor is the %TypedArray% intrinsic object.

15.13.6.3.2 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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
4. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Return *buffer*.

15.13.6.3.3 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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
4. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Let *size* be the value of *O*'s [[ByteLength]] internal data property.
7. Return *size*.

15.13.6.3.4 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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
4. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Let *offset* be the value of *O*'s [[ByteOffset]] internal data property.
7. Return *offset*.

15.13.6.3.5 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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
4. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.

Deleted: <#>15.13.6.3.2 *TypeArray*.BYTES_PER_ELEMENT¶
The initial value of *TypeArray*.BYTES_PER_ELEMENT the element size in bytes.¶
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.¶

Deleted: <#>15.13.6.4 Properties of the *TypeArrayTypedArray* Prototype Object¶

Deleted: each *TypeArrayTypedArray* prototype

Deleted: 15.2.4

Deleted: *TypedArray* prototype

Deleted: s

Deleted: are

Deleted: s

Deleted: They

Deleted: [[TypedArrayData]]

Deleted: .

Deleted: It's [[Class]] is "TypeArray".

Deleted: *TypeArray*

Deleted: *TypedArray*

Deleted: TypeArray

Deleted: standard built-in

Deleted: TypeArray

Deleted: constructor

Comment [AWB13212]: buffer needs to be an accessor both to comply with WebIDL requirements an to support the Kronos neutering strawman requirement

Deleted: <#>*TypedArray*.prototype.BYTES_PE ELEMENT¶

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: :

Deleted: [[TypedArrayData]]

Deleted: [[TypedArrayData]]

Comment [AWB13213]: buffer needs to be an

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: [[TypedArrayData]]

Deleted: [[TypedArrayData]]

Comment [AWB13214]: buffer needs to be an

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: [[TypedArrayData]]

Deleted: [[TypedArrayData]]

Comment [AWB13215]: buffer needs to be an

Deleted: *TypedArray*

Deleted: *TypedArray*

Deleted: [[TypedArrayData]]

Deleted: [[TypedArrayData]]

6. Let *length* be the value of *O*'s [[ArrayLength]] internal data property.
7. Return *length*.

15.13.6.3.6 %TypedArray%.prototype.set(array, offset = 0)

Set multiple values in this *TypedArray*, reading the values from the object *array*. The optional *offset* value indicates the first element index in this *TypedArray* where values are written. If omitted, it is assumed to be 0.

1. Assert: *typedArray* does not have a [[ViewedArrayBuffer]] internal data property. If it does, the definition in 15.13.6.6.8 applies.
2. Let *target* be the result of calling ToObject with the **this** value as its argument.
3. ReturnIfAbrupt(*target*).
4. If *target* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
5. Let *targetBuffer* be the value of *target*'s [[ViewedArrayBuffer]] internal data property.
6. If *targetBuffer* is **undefined**, then throw a **TypeError** exception.
7. Let *targetLength* be the value of *target*'s [[ArrayLength]] internal data property.
8. Let *targetOffset* be ToInteger (*offset*)
9. ReturnIfAbrupt(*targetOffset*).
10. If *targetOffset* < 0, then throw a **RangeError** exception.
11. Let *targetName* be the string value *target*'s [[TypedArrayName]] internal data property.
12. Let *targetElementSize* be the Number value of the Element Size value specified in Table 36 for *targetName*.
13. Let *targetType* be the string value of the Element Type value in Table 36 for *targetName*.
14. Let *targetByteOffset* be the value of *target*'s [[ByteOffset]] internal data property.
15. Let *src* be the result of ToObject(*array*).
16. ReturnIfAbrupt(*src*).
17. Let *srcLen* be the result of Get(*src*, `"length"`).
18. Let *numberLength* be ToNumber(*srcLen*).
19. Let *srcLength* be ToInteger(*numberLength*).
20. ReturnIfAbrupt(*srcLength*).
21. If *numberLength* ≠ *srcLength* or *srcLength* < 0, then throw a **TypeError** exception.
22. If *srcLength* + *targetOffset* > *targetLength*, then throw a **RangeError** exception.
23. Let *targetByteIndex* be *targetOffset* × *targetElementSize* + *targetByteOffset*.
24. Let *k* be 0.
25. Let *limit* be *targetByteIndex* + *targetElementSize* × min(*srcLength*, *targetLength* – *targetOffset*).
26. Repeat, while *targetByteIndex* < *limit*
    a. Let *Pk* be ToString(*k*).
    b. Let *kValue* be the result of Get(*src*, *Pk*).
    c. Let *kNumber* be ToNumber(*kValue*).
    d. ReturnIfAbrupt(*kNumber*).
    e. Perform SetValueInBuffer(*targetBuffer*, *k* × *targetElementSize*, *targetType*, *kNumber*).
    f. Set *k* to *k* + 1.
    g. Set *targetByteIndex* to *targetByteIndex* + *targetElementSize*.
27. Return **undefined**.

15.13.6.3.7 %TypedArray%.prototype.set(typedArray, offset = 0)

Set multiple values in this *TypedArray*, reading the values from the *typedArray* parameter 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 [[ViewedArrayBuffer]] internal data property. If it does not, the definition in 15.13.6.6.7 applies.
2. Let *target* be the result of calling ToObject with the **this** value as its argument.
3. ReturnIfAbrupt(*target*).
4. If *target* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
5. Let *targetBuffer* be the value of *target*'s [[ViewedArrayBuffer]] internal data property.
6. If *targetBuffer* is **undefined**, then throw a **TypeError** exception.
7. Let *targetLength* be the value of *target*'s [[ArrayLength]] internal data property.
8. Let *targetOffset* be ToInteger (*offset*)
9. ReturnIfAbrupt(*targetOffset*).

Deleted: *TypedArray*
Deleted: [[TypedArrayData]]
Deleted: [[TypedArrayData]]
Deleted: [[TypedArrayData]]
Deleted: Positive
Deleted: Uint32
Deleted: <#>Let *targetType* be the value of *target*'s [[TypedArrayElementKind]] internal da property.¶
Deleted: *targetType*
Deleted: <#>Let *srcLength* be ToPositiveInte Uint32(*srcLen*).¶ <#>ReturnIfAbrupt(*srcLength*).¶
Deleted: *data*
Deleted: *e*
Deleted: *element*
Deleted: *TypeArray*
Deleted: *TypedArray*
Deleted: array [
Deleted:]
Deleted: the
Deleted: *a*
Deleted: input
Deleted: ., *reading* input values from the arr
Deleted: the
Deleted: current array
Deleted: [[TypedArrayData]]
Deleted: [[TypedArrayData]]
Deleted: [[TypedArrayData]]
Deleted: <#>If this does not have class "TypeArray", throw a **TypeError** exception.¶
Deleted: *offsetIndex*
Deleted: Positive
Deleted: UInt32

10. If *targetOffset* < 0, then throw a **RangeError** exception.
11. Let *targetName* be the string value *target*'s [[TypedArrayName]] internal data property.
12. Let *targetType* be the string value of the Element Type value in Table 36 for *targetName*.
13. Let *targetElementSize* be the Number value of the Element Size value specified in Table 36 for *targetName*.
14. Let *targetByteOffset* be the value of *target*'s [[ByteOffset]] internal data property.
15. Let *srcBuffer* be the value of *typedArray*'s [[ViewedArrayBuffer]] internal data property.
16. If *srcBuffer* is **undefined**, then throw a **TypeError** exception.
17. Let *srcName* be the string value *typedArray*'s [[TypedArrayName]] internal data property.
18. Let *srcType* be the string value of the Element Type value in Table 36 for *srcName*.
19. Let *srcElementSize* be the Number value of the Element Size value specified in Table 36 for *srcName*.
20. Let *srcLength* be the value of *typedArray*'s [[ArrayLength]] internal data property.
21. Let *srcByteOffset* be the value of *typedArray*'s [[ByteOffset]] internal data property.
22. If *srcLength* + *targetOffset* > *targetLength*, then throw a **RangeError** exception.
23. If SameValue(*srcBuffer*, *targetBuffer*) is **true**, then
    a. Let *srcBuffer* be the result of calling CloneArrayBuffer(*srcBuffer*, *srcType*, *srcType*, *srcLength*).
24. Let *targetByteIndex* be *targetOffset* × *targetElementSize* + *targetByteOffset*.
25. Let *srcByteIndex* be *srcByteOffset*.
26. Let *limit* be *targetByteIndex* + *targetElementSize* × min(*srcLength*, *targetLength* – *targetOffset*).
27. Repeat, while *targetByteIndex* < *limit*
    a. Let *value* be the result of GetValueFromBuffer (*srcBuffer*, *srcByteIndex*, *srcType*).
    b. Let *status* be the result of SetValueInBuffer (*targetBuffer*, *targetByteIndex*, *targetType*, *value*).
    c. Set *srcByteIndex* to *srcByteIndex* + *srcElementSize*.Return
    d. Set *targetByteIndex* to *targetByteIndex* + *targetElementSize*.
28. Return **undefined**.

15.13.6.3.8 %TypedArray%.prototype.subarray(begin = 0, end = this.length)

Returns a new *TypedArray* object whose element types is the same as this *TypedArray* and whose ArrayBuffer is the same as the ArrayBuffer of this *TypedArray*, referencing the elements at *begin*, inclusive, up to *end*, exclusive. If either *begin* or *end* is negative, it refers to an index from the end of the array, as opposed to from the beginning.

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.
4. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Let *srcLength* be the value of *O*'s [[ArrayLength]] internal data property.
7. Let *beginInt* be ToInteger(*begin*)
8. ReturnIfAbrupt(*beginInt*).
9. If *beginInt* < 0, then let *beginInt* be *srcLength* + *beginInt*.
10. Let *beginIndex* be min(*srcLength*, max(0, *beginInt*)).
11. If *end* is **undefined**, then let *end* be *srcLength*.
12. Let *endInt* be ToInteger(*end*).
13. ReturnIfAbrupt(*endInt*).
14. If *endInt* < 0, then let *endInt* be *srcLength* + *endInt*.
15. Let *endIndex* be max(0,min(*srcLength*, *endInt*)).
16. If *endIndex* < *beginIndex*, then let *endIndex* be *beginIndex*.
17. Let *newLength* be *endIndex* - *beginIndex*.
18. Let *constructorName* be the string value *O*'s [[TypedArrayName]] internal data property.
19. Let *elementType* be the string value of the Element Type value in Table 36 for *constructorName*.
20. Let *elementSize* be the Number value of the Element Size value specified in Table 36 for *constructorName*.
26. Let *srcByteOffset* be the value of *O*'s [[ByteOffset]] internal data property.
27. Let *beginByteOffset* be *srcByteOffset* + *beginIndex* × *elementSize*.
28. Let *constructor* be the result of Get(*O*, `"constructor"`).
29. ReturnIfAbrupt(*constructor*).
30. If IsConstructor(*constructor*) is **false**, then throw a **TypeError** exception.
31. Let *argumentsList* be a List consisting of *buffer*, *beginByteOffset*, and *newLength*.
32. Return the result of calling the [[Construct]] internal method of *constructor* with *argumentsList* as the argument.

Deleted: the value of *target*'s [[TypedArrayElementKind
internal data property

Deleted: *targetType*

Deleted: <#>Let O be the result of calling ToObject(array).¶
<#>Let srcLength be the result Get(*O*, `"length"`)of call [[Get]] on O with property name "length".¶
<#>Let targetLength be the result of calling [[Get]] on (th with property name, `"length"`)."length"¶

Deleted: [[TypedArrayData]]

Deleted: the value of *typedArray*'s [[TypedArrayElementKind]] internal data property

Deleted: *srcType*

Deleted: offset

Deleted: ¶

Comment [AWB13216]: TODO

Deleted: *Data*

Deleted:

Comment [AWB13217]: TODO: not yhet updated fr wik

Deleted: <#>Let temp be a new TypeArray created as i a call to "new TypeArray(srcLength)"¶
<#>Let k be 0¶
<#>While *k* < *srcLength*¶
<#>Let *v* be the result of calling [[Get]] on(*src* with property name, toString(*k*)).¶
<#>Call [[Put]] on(*temp* , with arguments ToString(*k*), **and false**).¶
<#>Let *k* be *offset*.¶
<#>While k < targetLength¶
<#>Let *v* be the result of calling [[Get]] on(*temp* with property name, ToString(*k-offset*)).¶

Deleted: ***TypedArray***

Deleted: [

Deleted:]

Deleted: view

Deleted: of the

Deleted: store for

Deleted: [[TypedArrayData]]

Deleted: [[TypedArrayData]]

Deleted: <#>If this does not have class "TypeArray",

Deleted: Let *srcLength* be the result Get(this, `"length`

Deleted: of calling [[Get]] on this with property name

Deleted: 32

Deleted: 32

Deleted: if end was provided, else srcLength

Deleted: the value of *O*'s [[TypedArrayElementKind]]

Deleted: Table 36Table 35Table 35Table 36

Deleted: *elementType*

Deleted: return

Deleted: t

15.13.6.3.9 %TypedArray%.prototype.toString ( )

The value of the %TypedArray%.prototype.toString data property is the same built-in function object as the Array.prototype.toString method defined in 15.4.3.2.

Deleted: 15.4.3.2

15.13.6.3.10 %TypedArray%.prototype.toLocaleString ( )

The value of the %TypedArray%.prototype.toLocaleString data property is the same built-in function object as the Array.prototype.toLocaleString method defined in 15.4.3.3.

Comment [AWB16218]: Note that this applies ToUint32 to the length. In theory that a problem for TypedArrays but in practice is probably isn't for toLocaleString

Deleted: 15.4.3.3

15.13.6.3.11 %TypedArray%.prototype.join (separator)

15.13.6.3.12 %TypedArray%.prototype.reverse ( )

15.13.6.3.13 %TypedArray%.prototype.slice ( )

15.13.6.3.14 %TypedArray%.prototype.sort ( )

15.13.6.3.15 %TypedArray%.prototype.indexOf ( )

15.13.6.3.16 %TypedArray%.prototype.lastIndexOf ( )

15.13.6.3.17 %TypedArray%.prototype.every ( )

15.13.6.3.18 %TypedArray%.prototype.some ( )

15.13.6.3.19 %TypedArray%.prototype.forEach ( )

15.13.6.3.20 %TypedArray%.prototype.map ( )

15.13.6.3.21 %TypedArray%.prototype.filter ( )

15.13.6.3.22 %TypedArray%.prototype.reduce ( )

15.13.6.3.23 %TypedArray%.prototype.reduceRight ( )

15.13.6.3.24 %TypedArray%.prototype.find ( )

15.13.6.3.25 %TypedArray%.prototype.findIndex ( )

15.13.6.3.26 %TypedArray%.prototype.entries ( )

The following steps are taken:

4. Let *O* be the result of calling ToObject with the **this** value as its argument.
5. ReturnIfAbrupt(*O*).
6. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and `"key+value"`.

15.13.6.3.27 %TypedArray%.prototype.keys ( )

The following steps are taken:

4. Let *O* be the result of calling ToObject with the **this** value as its argument.
5. ReturnIfAbrupt(*O*).
6. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and `"key"`.

15.13.6.3.28 **%TypedArray%.prototype.values ( )**

The following steps are taken:

4. Let *O* be the result of calling ToObject with the **this** value as its argument.
5. ReturnIfAbrupt(*O*).
6. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and `"value"`.

15.13.6.3.29 **%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.

15.13.6.3.30 **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 result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. If *O* does not have a [[TypedArrayName]] internal data property throw a **TypeError** exception.
4. Let *name* be the value of *O*'s [[TypedArrayName]] internal data property.
5. Assert: *name* is a String value.
6. Return *name*.

This property has the attributes { [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.13.6.4 **The *TypedArray* Constructors**

Each of these *TypedArray* constructor objects has the structure described below, differing only in the name used as the constructor name instead of *TypedArray*, in Table 36.

When a *TypedArray* constructor is called as a function rather than as a constructor, it creates and initialises a new *TypedArray* object. Thus the function call ***`TypedArray`*** `(…)` is equivalent to the object creation expression `new` ***`TypedArray`*** `(…)` with the same arguments. However, if the **this** value value passed in the call is an Object with an [[ViewedArrayBuffer]] internal data property whose value is **undefined**, it initializes the **this** value using the argument values. This permits super invocation of the ***`TypedArray`*** constructors by ***`TypedArray`*** subclasses.

The ***`TypedArray`*** constructors are designed to be subclassable. They may be used as the value of an `extends` clause of a class declaration. Subclass constructors that intended to inherit the specified ***`TypedArray`*** behaviour must include a `super` call to the ***`TypedArray`*** constructor to initialise subclass instances.

15.13.6.5 **Properties of the *TypedArray* Constructors**

The value of the [[Prototype]] internal data property of each *TypedArray* constructor is the %TypedArray% intrinsic object (15.13.6.1).

Each *TypedArray* constructor has a [[TypedArrayConstructor]] internal data property whose String value is the the constructor name in the corresponding row in Table 36.

Besides a `length` property (whose value is 3), each *TypedArray* constructor has the following properties:

15.13.6.5.1 ***TypedArray*.prototype**

The initial value of *TypedArray*.prototype is the corresponding *TypedArray* prototype object (15.13.6.3).

Deleted: <#>Return a new TypeArray with the followin values for it's proeprtiesproperties:¶
<#> The length property of the newly constructed object is set to endIndex - beginIndex¶
<#> The byteLength property of the newly construc object is set to length multiplied by the size in bytes of Ty
<#> The buffer property of the newly constructed object is set to this.buffer.¶
<#> The byteOffset property of the newly constructe object is set to this.offset + beginIndex.¶

Deleted: <#>15.13.6.64.10 *TypedArray*.prototype.@@elementGet (index)
<#>Assert: Type(*index*) is number and *index* is an array index valueinteger.¶
<#>Let *O* be the result of calling ToObject with the **this** value as its argument.¶
<#>ReturnIfAbrupt(*O*).¶
<#>If *O* does not have a [[TypedArrayData]][[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.¶
<#>Let *buffer* be the value of *O*'s [[TypedArrayData]][[ViewedArrayBuffer]] internal data property.¶
<#>If *buffer* is **undefined**, then throw a **TypeError** exception.¶
<#>Let *length* be the value of *O*'s [[ArrayLength]] interna data property.¶
<#>Let *intIndex* be ToPositiveInteger Uint32(*index*).¶
<#>ReturnIfAbrupt(*index*).¶
<#>If *index* < 0 or *index intIndex* ≥ *length*, then return **undefined**.¶
<#>Let *offset* be the value of *O*'s [[ByteOffset]] internal d property.¶
<#>Let *elementSize* be the Number value of the Element S value specified in Table 36 for *TypedArray*.¶
<#>Let *indexedPosition* = (*index intIndex* × *elementSize*) *offset*.¶
<#>Let *elementType* be the value of *O*'s [[TypedArrayElementKind]] internal data property.¶
<#>Return the result of GetValueFromArrayBuffer(*buffer indexedPosition*, *elementType*).¶
<#>15.13.6.64.11 *TypedArray*.prototype.@@elementSet (index, value)¶
<#>Assert: Type(*index*) is number and *index* is an integer*index* is an array index value.¶
<#>Let *O* be the result of calling ToObject with the **this** value as its argument.¶
<#>ReturnIfAbrupt(*O*).¶
<#>If *O* does not have a [[TypedArrayData]][[ViewedArrayBuffer]] internal data property throw a **TypeError** exception.¶
<#>Let *buffer* be the value of *O*'s [[TypedArrayData]][[ViewedArrayBuffer]] internal data property.¶
<#>If *buffer* is **undefined**, then throw a **TypeError** exception.¶

Deleted: TypedArray.prototype. [@@toStringTag]¶
The initial value of the @@toStringTag property is the string value of the name TypedArray.¶
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.¶

Deleted: 15.13.6.1

Deleted: 15.13.6.3

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.13.6.6 Properties of *TypedArray* Prototype Objects

The value of the [[Prototype]] internal data property of a *TypedArray* prototype object is the standard built-in %TypedArrayPrototype% object (15.13.6.3). A *TypedArray* prototype object is an ordinary object. It does not have a [[ViewedArrayBuffer]] or or any other of the internal data properties that are specific to *TypedArray* instance objects.

15.13.6.6.1 *TypedArray*.prototype.constructor

The initial value of a *TypedArray*.prototype.constructor is the corresponding standard built-in *TypedArray* constructor.

15.13.6.6.2 *TypedArray*.prototype.BYTES_PER_ELEMENT

The value of *TypedArray*.prototype.BYTES_PER_ELEMENT is the Number value of the Element Size value specified in Table 36 for *TypedArray*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.13.6.7 Properties of *TypedArray* Instances

TypedArray instances are exotic Integer Indexed Delegation objects. Each *TypedArray* instances inherits properties from the corresponding *TypedArray* prototype object. Each *TypedArray* instances have the following internal data properties: [[ViewedArrayBuffer]], [[TypedArrayName]], [[ByteLength]], [[ByteOffset]], and [[ArrayLength]].

15.13.7 DataView Objects

15.13.7.1 Abstract Operations For DataView Objects

The abstract operation GetViewValue(view, requestIndex, isLittleEndian, type) used by functions on DataView instances is defined as follows:

1. Let *v* be ToObject(*view*)
2. ReturnIfAbrupt(*v*).
3. If *v* does not have [[ViewedArrayBuffer]] internal data property, then throw a **TypeError** exception.
4. Let *buffer* be the value of *v's* [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Let *numberIndex* be ToNumber(*requestIndex*)
7. Let *getIndex* be ToInteger(*numberIndex*).
8. ReturnIfAbrupt(*getIndex*).
9. If *numberIndex* ≠ *getIndex* or *getIndex* < 0, then throw a **RangeError** exception.
10. Let *isLittleEndian* be ToBoolean(*isLittleEndian*).
11. ReturnIfAbrupt(*isLittleEndean*).
12. Let *viewOffset* be the value of *v's* [[ByteOffset]] internal data property.
13. Let *viewSize* be the value of *v's* [[ByteLength]] internal data property.
14. Let *elementSize* be the Number value of the Element Size value specified in Table 36 for *type*.
15. If *getIndex* +*elementSize* > *viewSize*, then throw a **RangeError** exception.
16. Let *bufferIndex* be *getOffset*+*viewOffset*.
17. Return the result of the GetValueFromBuffer(*buffer*, *bufferIndex*, *type*, *isLittleEndean*).

The abstract operation SetViewValue(view, requestIndex, isLittleEndian, type, value) used by functions on DataView instances is defined as follows:

1. Let *v* be ToObject(*view*)
2. ReturnIfAbrupt(*v*).
3. If *v* does not have [[ViewedArrayBuffer]] internal data property, then throw a **TypeError** exception.

Deleted: 15.13.6.3

Deleted: After initialisation by a *TypedArray* constructor,

Deleted: [[TypedArrayData]]

Deleted: ElementKind

Deleted: <#>*TypeArray* instances inhe properties from the *TypeArray* prototyp object and their [[Class]] internal data property value is "TypeArray". *TypeArr* instances also have the following properties.¶
<#>15.13.6.5.1 [[DefineOwnProperty]] (desc, throw)¶
<#>TypeArray objects use a variation o the [[DefineOwnProperty]] internal meth used for other native ECMAScript objec (8.12.9).¶
<#>When the [[DefineOwnProperty]] internal method of A is called with property P, Property Descriptor Desc ar Boolean flag Throw, the following steps are taken:¶
<#> Let succeeded be the result of calling the default [[DefineOwnProperty internal method (8.12.9) on A passing P Desc, and Throw as arguments.¶
<#> If succeeded is false, return false
<#> If Desc contains a Value field, let newValue be Desc.Value¶
<#> Let convertedValue to ToType(newValue)¶
<#> Let index be ToUInt32(P)¶
<#> Call the SetValueInBuffer interna operation with arguments A.buffer.[[NativeBuffer]], A.byteOffset, index, convertedValue, and Type.¶
<#> Return true.¶
<#>The internal operation SetValueInBuffer takes five parameters, native buffer nativeBuffer, an integer byteOffset, an integer index, a value of type Type newValue, and a Type valueType. It operates as follows:¶
<#> Let size be the size in bytes of th type valueType.¶
<#> Let bytes be the array of bytes fr nativeBuffer between offset byteOffset+(index*size) and offset byteOffset+((index+1)* × size)-1 inclusi
<#> Let newValueBytes be the result converting newValue to an array of byte using the platform endianness.¶
<#> Set each byte of bytes from the corresponding byte of newValueBytes.¶
<#>15.13.6.5.2 [[GetOwnProperty]] (P)¶
<#>TypeArray objects use a variation o the [[GetOwnProperty]] internal method used for other native ECMAScript objec (8.12.1). This special internal method provides access to named properties corresponding to the individual index values of the TypeArray objects.¶
<#>When the [[GetOwnProperty]] inter method of A is called with property nam P, the following steps are taken:¶
<#> Let desc be the result of calling t default [[GetOwnProperty]] internal method (8.12.1) on A with argument P.¶

4. Let *buffer* be the value of *v's* [[ViewedArrayBuffer]] internal data property.
5. If *buffer* is **undefined**, then throw a **TypeError** exception.
6. Let *numberIndex* be ToNumber(*requestIndex*)
7. Let *getIndex* be ToInteger(*numberIndex*).
8. ReturnIfAbrupt(*getIndex*).
9. If *numberIndex* ≠ *getIndex* or *getIndex* < 0, then throw a **RangeError** exception.
10. Let *isLittleEndian* be ToBoolean(*isLittleEndian*).
11. ReturnIfAbrupt(*isLittleEndean*).
12. Let *viewOffset* be the value of *v's* [[ByteOffset]] internal data property.
13. Let *viewSize* be the value of *v's* [[ByteLength]] internal data property.
14. Let *elementSize* be the Number value of the Element Size value specified in Table 36 for *type*.
15. If *getIndex* +*elementSize* > *viewSize*, then throw a **RangeError** exception.
16. Let *bufferIndex* be *getOffset*+*viewOffset*.
17. Return the result of the SetValueInBuffer(*buffer*, *bufferIndex*, *type*, *value* *isLittleEndean*).

NOTE The algorithms for GetViewValue and SetViewValue are identical except for their final steps.

15.13.7.2 The DataView Constructor

When **`DataView`** is called as a function rather than as a constructor, it creates and initialises a new **`DataView`** object. Thus the function call **`DataView(…)`** is equivalent to the object creation expression **`new DataView(…)`** with the same arguments. However, if the **this** value value passed in the call is an Object with an [[ViewedArrayBuffer]] internal data property whose value is **undefined**, it initializes the **this** value using the argument values. This permits **`DataView`** to be used both as factory method and to perform constructor instance initialization.

The **`DataView`** constructor is designed to be subclassable. It may be used as the value of an **`extends`** clause of a class declaration. Subclass constructors that intended to inherit the specified **`ArrayBuffer`** behaviour must include a **`super`** call to the **`DataView`** constructor to initialise subclass instances.

15.13.7.2.1 DataView(buffer, byteOffset=0, byteLength=undefined)

`DataView` called with arguments *buffer*, *byteOffset*, and *length* performs the following steps:

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object or if *O* does not have an [[ViewedArrayBuffer]] internal data property or if the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then
   a. Let *F* be this function object.
   b. Let *argumentsList* be the *argumentsList* argument of the [[Call]] internal method that invoked *F*.
   c. Return the result of calling OrdinaryConstruct (*F*, *argumentsList*).
3. If Type(*buffer*) is not Object, then throw a **TypeError** exception.
4. If *buffer* does not have a [[ArrayBufferData]] internal data property, then throw a **TypeError** exception.
5. Let *numberOffset* be ToNumber(*byteOffset*).
6. Let *offset* be ToInteger(*numberOffset*).
7. ReturnIfAbrupt(*offset*).
8. If *numberOffset* ≠ *offset* or *offset* < 0, then throw a **RangeError** exception.
9. Let *bufferByteLength* be the value of *buffer's* [[ArrayBufferByteLength]] internal data property.
10. If *offset* ≥ *bufferByteLength*, then throw a **RangeError** exception.
11. If *byteLength* is **undefined**, then
    a. Let *viewByteLength* be *bufferByteLength* – *offset*.
12. Else,
    a. Let *numberLength* be ToNumber(*byteLength*).
    b. Let *viewLength* be ToInteger (*numberLength*).
    c. ReturnIfAbrupt(*viewLength*).
    d. If *numberLength* ≠ *viewLength* or *viewLength* < 0, then throw a **RangeError** exception.
    e. Let *viewByteLength* be *viewLength*.
    f. If *offset*+*viewByteLength* > *bufferByteLength*, then throw a **RangeError** exception.
13. If the value of *O's* [[ViewedArrayBuffer]] internal data property is not **undefined**, then throw a **TypeError** exception.

Deleted: <#>15.13.7.2 The DataView Construct
When DataView is called as part of a new expression,
a constructor: it initialises the newly created object.¶
<#>15.13.7.2.1 new DataView(buffer [, byteOffs byteLength]])¶
The [[Prototype]] internal data property of the newly constructed object is set to the original DataView prototype object, the one that is the initial value of DataView.prototype (15.13.3.3.1). The [[Class]] interna data property of the newly constructed object is set to "DataView". The [[Extensible]] internal data property of the newly constructed object is set to **true**.¶
The remaining proeprtiesproperties are set as follows:¶
<#> Let O be ToObject(buffer)¶
<#> If the [[Class]] internal data property of O is not "ArrayBuffer", raisethrow a **TypeError** exception.¶
<#> Let byteOffset be the result of calling ToUInt32 on byteOffset, if provided, or else 0.¶
<#> Let bufferLength be the result of calling [[Get]] on(with property name, **`""byteLength""`**).¶
<#> Let byteLength be the result of calling ToUInt32 or byteLength, if provided, or else bufferLength – byteOffset.
<#> If byteOffset + byteLength is greater than bufferLength, throw raise a **RangeError** exception.¶
<#> The byteLength property of the newly constructed object is set to byteLength.¶
<#> The buffer property of the newly constructed objec set to O.¶
<#> The byteOffset property of the newly constructed object is set to byteOffset.¶

14. Set *O's* [[ViewedArrayBuffer]] to *buffer*.
15. Set *O*'s [[ByteLength]] internal data property to *newByteLength*.
16. Set *O*'s [[ByteOffset]] internal data property to *offset*.
17. Return *O*.

15.13.7.2.2 new DataView(... argumentsList)

`DataView` called as part of a new expression it performs the following steps:

4. Let *F* be the function object on which the **`new`** operator was applied.
5. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that invoked by the **`new`** operator.
6. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If `DataView` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.13.7.3 Properties of the DataView Constructor

The value of the [[Prototype]] internal data property of the `DataView` constructor is the `Function` prototype object (15.3.3).

Besides the internal properties and the length property (whose value is 3), the DataView constructor has the following properties:

15.13.7.3.1 DataView.prototype

The initial value of `DataView.prototype` is the DataView prototype object (15.13.7.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.13.7.3.2 DataView [@@create] ( )

The @@create method of a DataView function object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%DataViewPrototype%"`, ([[ViewedArrayBuffer]] , [[ByteLength]], [[ByteOffset]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.13.7.4 Properties of the DataView Prototype Object

The value of the [[Prototype]] internal data property of the DataView prototype object is the standard built-in Object prototype object (15.2.4). The DataView prototype object is an ordinary object. It does not have a [[ViewedArrayBuffer]], [[ByteLength]], or [[ByteOffset]] internal data property.

15.13.7.4.1 DataView.prototype.constructor

The initial value of `DataView.prototype.constructor` is the standard built-in DataView constructor.

15.13.7.4.2 DataView.prototype.getInt8(byteOffset)

When the `getInt8` method is called with argument *byteOffset* the following steps are taken:

1. Let *v* be the this value.
2. Return the result of GetViewValue(*v*, *byteOffset*, **undefined**, `"Int8"`).

Deleted: 15.3.3

Deleted: 15.13.7.4

Deleted: The [[Class]] internal data property the newly constructed object is set to "Object The [[Extensible]] internal data property of the newly constructed object is set to **true**.

Deleted: The internalabstract operation GetValue(byteOffset, isLittleEndian, type) use by functions on DataView instances is define as follows:¶
<#> Let byteOffsetInt be ToUInt32(byteOffs
<#> Let totalOffset be byteOffsetInt plus the result of calling [[Get]] on(this with parameter, "`"byteOffset"`").¶
<#> Let byteLength be the result of calling [[Get]] on(this with parameter, "`"byteLength"`").¶
<#> If totalOffset >>= byteLength, throw rai **RangeError** exception.¶
<#> Let value be the result of calling the GetValueFromBuffer internalabstract operation (2.5.2) with arguments this.buffer.[[NativeBuffe totalOffset, 0 and type.¶
<#> Return value¶
The internalabstract operation SetValue(byteOffset, isLittleEndian, type, valu used by functions on DataView instances is defined as follows:¶
<#> Let byteOffsetInt be ToUInt32(byteOffs
<#> Let totalOffset be byteOffsetInt plus the result of calling [[Get(this, `"byteOffset"`).Get]] on this with parameter "byteOffset"¶
<#> Let byteLength be the result of Get(this, `"byteLength"`).calling [[Get]] on this with parameter "byteLength"¶
<#> If totalOffset >>= byteLength, raisethrow **RangeError** exception.¶
<#> Let value be the result of calling the SetValueInBuffer internalabstract operation (2.5 with arguments this.buffer.[[NativeBuffer]], totalOffset, 0, value and type.¶
<#> Return value¶
15.13.7.4.1

Deleted: Gets the Int8 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶

Deleted:

Deleted: **true**

15.13.7.4.3 **DataView.prototype.getUint8(byteOffset)**

When the `getUint8` method is called with argument *byteOffset* the following steps are taken:

1. Let *v* be the **this** value.
2. Return the result of GetViewValue(*v*, *byteOffset*, **undefined**, **`"Uint8"`**).

15.13.7.4.4 **DataView.prototype.getInt16(byteOffset, littleEndian=false)**

When the `getInt16` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, **`"Int16"`**).

15.13.7.4.5 **DataView.prototype.getUint16(byteOffset, littleEndian=false)**

When the `getUint16` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, **`"Uint16"`**).

15.13.7.4.6 **DataView.prototype.getInt32(byteOffset, littleEndian=false)**

When the `getInt32` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **undefined**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, **`"Int32"`**).

15.13.7.4.7 **DataView.prototype.getUint32(byteOffset, littleEndian=false)**

When the `getUint32` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, **`"Uint32"`**).

15.13.7.4.8 **DataView.prototype.getFloat32(byteOffset, littleEndian=false)**

When the `getFloat32` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, **`"Float32"`**).

15.13.7.4.9 **DataView.prototype.getFloat64(byteOffset, littleEndian=false)**

When the `getFloat64` method is called with argument *byteOffset* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.

Deleted: Gets the UInt8 value at offset byteOffset in th
DataView.¶
<#> Let O be ToObject(this)¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, **true**, UInt8)¶
15.13.7.4.4

Deleted: Gets the Int16 value at offset byteOffset in th
DataView, using the provided endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEndian, Int16)
15.13.7.4.5

Deleted: Gets the Uint16 value at offset byteOffset in
DataView, using the provided endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEndian, Uint16

Deleted: Gets the Int32 value at offset byteOffset in th
DataView, using the provided endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEndian, Int32)
15.13.7.4.7

Deleted: Gets the Uint32 value at offset byteOffset in
DataView, using the provided endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEndian, Uint32
15.13.7.4.8

Deleted: Gets the Float32 value at offset byteOffset in
the DataView, using the provided endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEndian, Float3
15.13.7.4.9

2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of GetViewValue(*v*, *byteOffset*, *littleEndean*, `"Float64"`).

15.13.7.4.10 DataView.prototype.setInt8(byteOffset, value)

When the `setInt8` method is called with arguments *byteOffset* and *value* the following steps are taken:

1. Let *v* be the this value.
2. Return the result of SetViewValue(*v*, *byteOffset*, **undefined**, `"Int8"`, *value*).

15.13.7.4.11 DataView.prototype.setUint8(byteOffset, value)

When the `setUint8` method is called with arguments *byteOffset* and *value* the following steps are taken:

1. Let *v* be the **this** value.
2. Return the result of SetViewValue(*v*, *byteOffset*, **undefined**, `"Uint8"`, *value*).

15.13.7.4.12 DataView.prototype.setInt16(byteOffset, value, littleEndian=false)

When the `setInt16` method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, `"Int16"`, *value*).

15.13.7.4.13 DataView.prototype.setUint16(byteOffset, value, littleEndian=false)

When the `setUint16` method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, `"Uint16"`, *value*).

15.13.7.4.14 DataView.prototype.setInt32(byteOffset, value, littleEndian=false)

When the `setInt32` method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, `"Int32"`, *value*).

15.13.7.4.15 DataView.prototype.setUint32(byteOffset, value, littleEndian=false)

When the `setUint32` method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, `"Uint32"`, *value*).

15.13.7.4.16 DataView.prototype.setFloat32(byteOffset, value, littleEndian=false)

When the `setFloat32` method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

Deleted: Gets the Float64 value at offset byteOffset in the DataView, using the provide endianness.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶
<#> Return GetValue(byteOffset, isLittleEnd Float64)¶
15.13.7.4.10

Deleted: Sets the Int8 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶
<#> Return GSetValue(byteOffset, **true**, Int8 ToInt8(value))¶
15.13.7.4.11

Deleted: Sets the Uint8 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶
<#> Return GSetValue(byteOffset, **true**, Uin ToUint8(value))¶
15.13.7.4.12

Deleted: Sets the Int16 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶
<#> Return GSetValue(byteOffset, isLittleEndian, Int16, ToInt16(value))¶
15.13.7.4.13

Deleted: Sets the Uint16 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶

Deleted: Sets the Int32 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶

Deleted: Sets the Uint32 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of not "DataView", throw raise a **TypeError** exception.¶

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, **"Float32"**, *value*).

15.13.7.4.17 DataView.prototype.setFloat64(byteOffset, value, littleEndian=false)

When the **setFloat64** method is called with arguments *byteOffset* and *value* and optional argument *littleEndean* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndean* is not preset, then let *littleEndian* be **false**.
3. Return the result of SetViewValue(*v*, *byteOffset*, *littleEndian*, **"Float64"**, *value*).

15.13.7.4.18 DataView.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value **"DataView"**.

15.13.7.5 Properties of DataView Instances

DataView instances are ordinary objects that inherit properties from the DataView prototype object. Map instances each have a [[ViewedArrayBuffer]], [[ByteLength]], and [[ByteOffset]] internal data properties. DataView instances also have the following properties.

15.13.7.5.1 byteLength

The value of the byteLength property is the length of the DataView object. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

15.13.7.5.2 buffer

The value of the buffer property is the ArrayBuffer accessed by the DataView object. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

15.13.7.5.3 byteOffset

The value of the byteOffset property is the length of the DataView object. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

15.14 Map Objects

Map objects are collections of key/value pairs where both the keys and values may be arbitrary ECMAScript language values. A distinct key value may only occur in one key/value pair within the Map's collection. Distinct key values as discriminated using the a comparision algorithm that is selected when the Map is created.

A Map object can iterate its elements in insertion order. Map object must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Map objects specification is only intended to describe the required observable semantics of Map objects. It is not intended to be a viable implementation model.

15.14.1 The Map Constructor

The Map constructor is the %Map% instrinsic object and the initial value of the **Map** property of the global object. When **Map** is called as a function rather than as a constructor, it initialises its **this** value with the internal state necessary to support the **Map.prototype** internal methods.

Deleted: Sets the Float32 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GSetValue(byteOffset, isLittleEndian, Float ToFloat32(value))¶
15.13.7.4.17

Deleted: Uint16

Deleted: Sets the Float64 value at offset byteOffset in the DataView.¶
<#> Let O be ToObject(this)¶
<#> Let isLittleEndian be ToBoolean(littleEndian) if provided, else **false**¶
<#> If the [[Class]] internal data property of O is not "DataView", throw raise a **TypeError** exception.¶
<#> Return GSetValue(byteOffset, isLittleEndian, Float ToFloat64(value))¶
15.13.7.5

Deleted: and their [[Class]] internal data property valu is "DataView".

Deleted: ,

Deleted: which was fixed at creation.

Deleted: length

Deleted: of

Deleted: ,

Deleted: which was fixed at creation.

Deleted: ,

Deleted: which was fixed at creation.

Deleted: e

Deleted: also

Deleted: <#>15.14.1 Abstract Operations For M Objects¶
<#>15.14.1.1 MapInitialzsation¶
The abstract operation MapInitialzsation with argumen *object* and *iterable* is used to initialize an object as a m It performs the following steps:¶
<#>If Type(*obj*) is not Object, throw a **TypeError** exception.¶
<#>If *obj* alreadydoes not have has a [[MapData]] interna data property, throw a **TypeError** exception.¶
<#>If the result of calling the [[GetIsExtensible]] internal propertymethod of *obj* is **false**, throw a **TypeError** excepti
<#>If *iterable* is not **undefined**, then¶
<#>Let *iterable* be ToObject(*iterable*).¶
<#>ReturnIfAbrupt(*iterable*)¶
<#>Let *iterator* be the intrinsic symbol @@iterator.¶
<#>Let *itr* be the result of calling the Invoke(*iterableo* abstraction operation with *iterator*, *obj*, and an empty Li as arguments).¶
<#>ReturnIfAbrupt(*itr*).¶
<#>Let *adder* be the result of calling the [[Get]] interna method of(*obj* with argument, **"set"**).¶
<#>ReturnIfAbrupt(*adderaddr*).¶

Deleted: Called as a Function

Deleted: ize

Deleted:

The **`Map`** constructor is designed to be subclassable. It may be used as the value in an **`extends`** clause of a class definition. Subclass constructors that intend to inherit the specified **`Map`** behaviour must include a **`super`** call to **`Map`**.

15.14.1.1 Map (iterable = undefined , comparator = undefined)

When the **`Map`** function is called with optional arguments *iterable* and *comparator* the following steps are taken:

1. Let *map* be the **this** value.
2. If Type(*map*) is not Object then, throw a **TypeError** exception.
3. If *map* does not have a [[MapData]] internal data property, then throw a **TypeError** exception.
4. If *map's* [[MapData]] internal data property is not **undefined**, then throw a **TypeError** exception.
5. If *iterable* is not present, let *iterable* be **undefined**.
6. If *iterable* is either **undefined** or **null**, then let *iter* be **undefined**.
7. Else,
   a. Let *hasValues* be the result of HasProperty(*iterable*, **`"entries"`**).
   b. ReturnIfAbrupt(*hasValues*).
   c. If *hasValues* is **true**, then
      i. Let *iter* be the result of Invoke(*iterable*, **`"entries"`**).
   d. Else,
      i. Let *iter* be the result of GetIterator(*iterable*).
   e. ReturnIfAbrupt(*iter*).
   f. Let *adder* be the result of Get(*map*, **`"set"`**).
   g. ReturnIfAbrupt(*adder*).
   h. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
8. If *comparator* is not **undefined**, then
   a. If *comparator* is not **`"is"`**, then throw a **RangeError** Exception.
9. Set *map's* [[MapData]] internal data property to a new empty List.
10. Set *map's* [[MapComparator]] internal data property to *comparator*.
11. If *iter* is **undefined**, then return *map*.
12. Repeat
    a. Let *next* be the result of IteratorNext(*iter*).
    b. ReturnIfAbrupt(*next*).
    c. Let *done* be IteratorComplete(*next*).
    d. ReturnIfAbrupt(*done*).
    e. If *done* is **true**, then return NormalCompletion(*map*).
    f. Let *nextItem* be IteratorValue(*next*).
    g. ReturnIfAbrupt(*nextItem*).
    h. Let *k* be the result of Get(*nextItem*, **`"0"`**).
    i. ReturnIfAbrupt(*k*).
    j. Let *v* be the result of Get(*nextItem*, **`"1"`**).
    k. ReturnIfAbrupt(*v*).
    l. Let *status* be the result of calling the [[Call]] internal method of *adder* with *map* as *thisArgument* and a List whose elements are *k* and *v* as *argumentsList*.
    m. ReturnIfAbrupt(*status*).

NOTE If the parameter *iterable* is present, it is expected to be an object that implements either an **`entries`** method or an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as an Map key and whose second element is the value to associate with that key.

15.14.1.2 new Map (... argumentsList)

When **`Map`** is called as part of a **`new`** expression it is a constructor: it initialises a newly created object.

`Map` called as part of a new expression with argument list *argumentsList* performs the following steps:

1. Let *F* be the Map function object on which the **`new`** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **`new`** operator.

Deleted: This premitspermits super invocati of the **`Map`** constructor by **`Map`** subclasses.

Deleted: [ ]

Deleted: *m*

Deleted: <#>If *m* is **undefined** or the intrinsi %MapPrototype%¶
<#>Let *map* be the result of the abstract operation ObjectCreate (15.2) with the intrins %MapPrototype% as the argument.¶
<#>Else¶

Deleted: <#>Let *map* be the result of ToObject(*m*).¶
<#>ReturnIfAbrupt(*map*).¶

Deleted: n

Deleted: *itr*

Deleted: *itr*

Deleted: *obj*

Deleted: <#>Let *iterator* be the well-known symbol @@iterator symbol.¶

Deleted: *itr*

Deleted: Invoke

Deleted: , *iterator*).

Deleted: *itr*

Deleted: <#>ReturnIfAbrupt(*itr*).¶

Deleted: Invoke

Deleted: , **`"next"`**

Deleted: <#>If Type(*next*) is not Object, t throw a **TypeError** exception.¶

Deleted: IteratorComplete(*next*)

Deleted: <#>Let *next* be ToObject(*next*).¶
<#>ReturnIfAbrupt(*next*).¶

Deleted: *next*

Deleted: *next*

Comment [AWB12219]: Note that using a method call for inserting pairs during initialization provides allows subclasses to be more expressive.

Deleted: <#>Let *status* be the result of MapInitializsation with *map* and *iterable* as arguments.¶

Deleted: <#>Return *map*.¶

Deleted: <#>The Map Constructor¶
When **`Map`** is called as part of a **`new`** expressi it is a constructor: it initialises thea newly created object. ¶
The **`Map`** constructor is designed to be subclassable. It may be used as the value of **`extends`** clause of a class declaration.

Deleted: ¶

Deleted: iterable = [ ]undefined

Deleted: args

Deleted: *args*

Deleted: be the be the

3. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If Map is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.14.2 Properties of the Map Constructor

The value of the [[Prototype]] internal data property of the Map constructor is the Function prototype object (15.3.3).

Besides the `length` property (whose value is **0**), the Map constructor has the following property:

15.14.2.1 Map.prototype

The initial value of `Map.prototype` is the Map prototype object (15.14.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.14.2.2 Map[@@create] ( )

The @@create method of a Map function object *F* performs the following steps:

4. Let *F* be the **this** value.
5. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%MapPrototype%"`, ([[MapData]] , [[MapComparator]])).
6. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.14.3 Properties of the Map Prototype Object

The value of the [[Prototype]] internal data property of the Map prototype object is the standard built-in Object prototype object (15.2.4). The Map prototype object is an ordinary object. It does not have a [[MapData]] or a [[MapComparator]] internal data property.

15.14.3.1 Map.prototype.constructor

The initial value of `Map.prototype.constructor` is the built-in `Map` constructor.

15.14.3.2 Map.prototype.clear ()

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
6. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
   a. Set *p*.[[key]] to empty.
   b. Set *p*.[[value]] to empty.
7. Return **undefined**.

15.14.3.3 Map.prototype.delete (key)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.

Deleted: with arguments

Deleted: and

Deleted: a

Deleted: <#>Let *map* be the result of the abstract operat ObjectCreate (15.2) with the intrinsic %MapPrototype% as the argument.¶
<#>If *iterable* is not present, let *iterable* be **undefined**.¶
<#>Let *status* be the result of MapInitialization with *map* and *iterable* as arguments.¶
<#>ReturnIfAbrupt(*status*).¶
<#>Return *map*.¶

Deleted: NOTE If the parameter iterable is present, i expected to be an object that implements an @@iterato method that returns an iterator object that produces tw element array-like objects whose first element is a valu that will be used as an Map key and whose second element is the value to associate with that key.¶

Deleted: 15.3.4

Deleted: internal properties and the

Deleted: ,

Deleted: <#>Add a [[MapData]] internal data property t *obj*.¶
<#>Set *obj*'s [[MapData]] internal data property to a new empty List.¶

Deleted: false

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

Deleted: TypeError

Deleted: <#>Set the value of *M*'s [[MapData]] internal data property to a new empty List.¶

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. If *M's* [[MapComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
6. Else, let *same* be the abstract operation SameValue.
7. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
8. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If *same*(*p*.[[key]], *key*), then
        i. Set *p*.[[key]] to empty.
        ii. Set *p*.[[value]] to empty.
        iii. Return **true**.
9. Return **false**.

NOTE The value **empty** is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

15.14.3.4 Map.prototype.forEach (callbackfn , thisArg = undefined)

callbackfn should be a function that accepts three arguments. **forEach** calls *callbackfn* once for each key/value pair present in the map object, in key insertion order. *callbackfn* is called only for keys of the map which actually exist; it is not called for keys that have been deleted from the map.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

NOTE If *callbackfn* is an Arrow Function, **this** was lexically bound when the function was created so *thisArg* will have no effect.

callbackfn is called with three arguments: the 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*.

NOTE Each key is visited only once with the value that is current at the time of the visit. If the value associated with a key is modified after it has been visited, it is not re-visited. Keys that are deleted after the call to **forEach** begins and before being visited are not visited. New keys added, after the call to **forEach** begins are visited.

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
6. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
7. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
8. Repeat for each Record {[[key]], [[value]]} *e* that is an element of *entries*, in original key insertion order
    a. If *e*.[[key]] is not empty, then
        i. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *e*.[[value]], *e*.[[key]], and *M* as *argumentsList*.
        ii. ReturnIfAbrupt(*funcResult*).
9. Return **undefined**.

The **length** property of the **forEach** method is **1**.

15.14.3.5 Map.prototype.get (key)

The following steps are taken:

Deleted: TypeError
Deleted: *S*
Deleted: Value
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: TypeError
Deleted: the **this** value
Deleted: argument
Deleted: l

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
6. If *M's* [[MapComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If *same*(*p*.[[key]], *key*), then return *p*.[[value]]
9. Return **undefined**.

15.14.3.6 Map.prototype.has (key)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
6. If *M's* [[MapComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If *same*(*p*.[[key]], *key*), then return **true**.
9. Return **false**.

15.14.3.7 Map.prototype.entries ( )

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and `"key+value"`.

15.14.3.8 Map.prototype.keys ( )

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and `"key"`.

15.14.3.9 Map.prototype.set (key , value)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
6. If *M's* [[MapComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If *same*(*p*.[[key]], *key*), then
        i. Set *p*.[[value]] to *value*.

Deleted: result of calling ToObject with the
Deleted: the as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: TypeError
Deleted: *S*
Deleted: Value
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: TypeError
Deleted: *S*
Deleted: Value
Comment [AWB14220]: Need to move after delete
Deleted: items
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: *O*
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: *O*
Deleted: the result of calling ToObject with
Deleted: as its argument
Deleted: ReturnIfAbrupt(*M*)
Deleted: TypeError
Deleted: TypeError
Deleted: *S*
Deleted: Value

ii. Return *M*.

9. Let *p* be the Record {[[key]]: *key*, [[value]]: *value*}
10. Append *p* as the last element of *entries*.
11. Return *M*.

15.14.3.10 get Map.prototype.size

Map.prototype.size is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[MapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
6. Let *count* be 0.
7. For each Record {[[key]], [[value]]} *p* that is an element of *entries*
   a. If *p*.[[key]] is not empty then
      i. Set *count* to *count*+1.
8. Return *count*.

15.14.3.11 Map.prototype.values ( )

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and **"value"**.

15.14.3.12 Map.prototype.@@iterator ( )

The initial value of the @@iterator property is the same function object as the initial value of the **entries** property.

15.14.3.13 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** }.

15.14.4 Properties of Map Instances

Map instances are ordinary objects that inherit properties from the Map prototype. Map instances also have a [[MapData]] internal data property and a [[MapComparator]] internal data property.

15.14.5 Map Iterator Object Structure

A Map Iterator is an object, with the structure defined below, that represent 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.

15.14.5.1 CreateMapIterator Abstract Operation

Several methods of Map objects return Iterator objects. The abstract operation CreateMapIterator with arguments *map* and *kind* is used to create such iterator objects. It performs the following steps:

1. Let *M* be the result of calling ToObject(*map*).
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.

Deleted: undefined
Deleted: undefined
Deleted: (key)
Deleted: The
Deleted: are taken
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: TypeError
Deleted: I s
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*M*)
Deleted: *O*
Comment [AWB10221]: This is how we identify a property whose key is a built-in Symbol
Deleted: items
Comment [AWB10222]: Do we really wan do this sort of method sharing. It has a bad smell.
Deleted: .
Deleted: ( )
Deleted: After initializsation by the Map constructor,
Deleted:
Deleted: s
Deleted: interator
Deleted: and

4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Let *iterator* be the result of ObjectCreate(%MapIteratorPrototype%, ([[Map]], [[MapNextIndex]], [[MapIterationKind]])).
6. Set *iterator's* [[Map]] internal data property to *M*.
7. Set *iterator's* [[MapNextIndex]] internal data property to 0.
8. Set *iterator's* [[MapIterationKind]] internal data property to *kind*.
9. Return *iterator*.

15.14.5.2 The Map Iterator Prototype

All Map Iterator Objects inherit properties from a common Map Iterator Prototype object. The [[Prototype]] internal data property of the Map Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Map Iterator Prototype as the following properties:

15.14.5.2.1 *MapIterator*.prototype.constructor

15.14.5.2.2 *MapIterator*.prototype.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 properties of a Map Iterator Instance (15.14.7.1.2), throw a **TypeError** exception.
4. Let *m* be the value of the [[Map]] internal data property of *O*.
5. Let *index* be the value of the [[MapNextIndex]] internal data property of *O*.
6. Let *itemKind* be the value of the [[MapIterationKind]] internal data property of *O*.
7. Assert: *m* has a [[MapData]] internal data property and *m* has been initialised so the value of [[MapData]] is not **undefined**.
8. Let *entries* be the List that is the value of the [[MapData]] internal data property of *m*.
9. Repeat while *index* is less than the total number of element of *entries*. The number of elements must be redetermined each time this method is evaluated.
   a. Let *e* be the Record {[[key]], [[value]]} at 0-origined insertion position *index* of *entries*.
   b. Set *index* to *index*+1;
   c. Set the [[MapNextIndex]] internal data property of *O* to *index*.
   d. If *e*.[[key]] is not empty, then
      i. If *itemKind* is `"key"` then, let *result* be *e*.[[key]].
      ii. Else if *itemKind* is `"value"` then, let *result* be *e*.[[value]].
      iii. Else,
         1. Assert: *itemKind* is `"key+value"`.
         2. Let *result* be the result of the abstract operation ArrayCreate with argument 2.
         3. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
         4. Call CreateOwnDataProperty(*result*, `"0"`, *e*.[[key]]).
         5. Call CreateOwnDataProperty(*result*, `"1"`, *e*.[[value]]).
      iv. Return CreateItrResultObject(*result*, **false**).
10. Return CreateItrResultObject(**undefined**, **true**).

15.14.5.2.3 *MapIterator*.prototype [@@iterator] ( )

The following steps are taken:

1. Return the **this** value.

15.14.5.2.4 *MapIterator*.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Map Iterator"`.

Deleted: *itr*
Deleted: the abstract operation
Deleted: with the intrinsic object
Deleted: as its argument
Deleted: Add a
Deleted: *itr*
Deleted: *itr* with value
Deleted: *itr*
Deleted: Add a
Deleted: *itr* with value
Deleted: *itr*
Deleted: Add a
Deleted: of *itr* with
Deleted: value
Deleted: *itr*
Deleted: s
Comment [AWB10223]: TODO: need to decide wh to use for a constructor for these sort of objects. Probably we should try to consistently follow the "class model" wherever we can.
Deleted: z
Deleted: Object
Deleted: the [[DefineOwnProperty]] internal method of
Deleted: with arguments
Deleted: Property Descriptor {[[Value]]:
Deleted: , [[Writable]]: **true**, [[Enumerable]]: tr [[Configurable]]: **true**},)
Deleted: and **false**
Deleted: the [[DefineOwnProperty]] internal method of
Deleted:
Deleted: with arguments
Deleted: Property Descriptor {[[Value]]:
Deleted: , [[Writable]]: **true**, [[Enumerable]]: tr [[Configurable]]: **true**}
Deleted: , and **false**
Deleted: Completion {[[type]]: throw, [[value]]: %StopIteration%, [[target]]: empty}
Deleted: .
Deleted: .@@

15.14.5.3 **Properties of Map Iterator Instances**

Map Iterator instances are ordinary objects that inherit properties from the Map Iterator prototype (the intrinsic, %MapIteratorPrototype%.) Map Iterator instances are initially created with the internal properties described in Table 37.

Table 37 — Internal Data Properties of Map Iterator Instances

Internal Data Property Name	Description
[[Map]]	The Map object that is being iterated.
[[MapNextIndex]]	The integer index of the next Map data element to be examined by this iteration.
[[MapIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: **"key"**, **"value"**, **"key+value"**.

15.15 WeakMap Objects

WeakMap objects are collections of key/value pairs where the keys are ECMAScript objects and values may be arbitrary ECMAScript language values. A WeakMap may be queried to see if it contains an key/value pair with a specific key, but no mechanisms is provided for enumerating the objects it holds as keys. If an object that is being used as the key of a WeakMap key/value pair is only reachable by following a chain of references that start within that WeakMap, then that key/value pair is inaccessible and is automatically removed from the WeakMap. WeakMap implementations must detect and remove such key/value pairs and any associated resources.

An implementation may impose an arbitrarily determined latency between the time a key/value pair of a WeakMap becomes inaccessible and the time when the key/value pair is removed from the WeakMap. If this latency was observable to ECMAScript program, it would be a source of indeterminacy that could impact program execution. For that reason, an ECMAScript implementation must not provide any means to observe a key of a WeakMap that does not require the observer to present the observed key.

WeakMap objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of key/value pairs in the collection. The data structure used in this WeakMap objects specification are only intended to describe the required observable semantics of WeakMap objects. It is not intended to be a viable implementation model.

NOTE WeakMap and WeakSets are intended to provide mechanisms for dynamically associating state with an object in a manner that does not "leak" memory resources if, in the absence of the WeakMap or WeakSet, the object otherwise became inaccessible and subject to resource reclamation by the implementation's garbage collection mechanisms. Achieving this characteristic requires coordination between the WeakMap or WeakSet implementation and the garbage collector. The following references describe mechanism that may be useful to implementations of WeakMap and WeakSets:

Barry Hayes. 1997. Ephemerons: a new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97)*, A. Michael Berman (Ed.). ACM, New York, NY, USA, 176-183. http://doi.acm.org/10.1145/263698.263733

Alexandra Barros, Roberto Ierusalimschy, Eliminating Cycles in Weak Tables. Journal of Universal Computer Science - J.UCS , vol. 14, no. 21, pp. 3481-3497, 2008. http://www.jucs.org/jucs_14_21/eliminating_cycles_in_weak

Deleted: following

Deleted: querried

Deleted: specifiec

Deleted: mechansims

Deleted:

Deleted: s

Deleted: should

Deleted: make

Deleted: m

Deleted: m

Deleted: directly test for the presensepreser of any specific

Deleted: value

Deleted: s

Deleted: a

Deleted: the object would

Deleted:

Deleted: become

Deleted: reclaimation

Deleted: ion

Deleted: s

15.15.1 The WeakMap Constructor

The WeakMap constructor is the %WeakMap% instrinsic object and the initial value of the `WeakMap` property of the global object. When `WeakMap` is called as a function rather than as a constructor, it initialises its **this** value with the internal state necessary to support the `WeakMap.prototype` methods.

The `WeakMap` constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `WeakMap` behaviour must include a `super` call to `WeakMap`.

15.15.1.1 WeakMap (iterable = undefined)

When the `WeakMap` function is called with optional argument *iterable* the following steps are taken:

1. Let *map* be the **this** value.
2. If Type(*map*) is not Object then, throw a **TypeError** exception.
3. If *map* does not have a [[WeakMapData]] internal data property, then throw a **TypeError** exception.
4. If *map's* [[WeakMapData]] internal data property is not **undefined**, then throw a **TypeError** exception.
5. If *iterable* is not present, let *iterable* be **undefined**.
6. If *iterable* is either **undefined** or **null**, then let *iter* be **undefined**.
7. Else,
    a. Let *hasValues* be the result of HasProperty(*iterable*, `"entries"`).
    b. ReturnIfAbrupt(*hasValues*).
    c. If *hasValues* is **true**, then
        i. Let *iter* be the result of Invoke(*iterable*, `"entries"`).
    d. Else,
        i. Let *iter* be the result of GetIterator(*iterable*).
    e. ReturnIfAbrupt(*iter*).
    f. Let *adder* be the result of Get(*map*, `"set"`).
    g. ReturnIfAbrupt(*adder*).
    h. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
8. Set *map's* [[WeakMapData]] internal data property to a new empty List.
9. If *iter* is **undefined**, then return *map*.
10. Repeat
    a. Let *next* be the result of IteratorNext(*iter*).
    b. ReturnIfAbrupt(*next*).
    c. Let *done* be IteratorComplete(*next*).
    d. ReturnIfAbrupt(*done*).
    e. If *done* is **true**, then return NormalCompletion(*map*).
    f. Let *nextValue* be IteratorValue(*next*).
    g. ReturnIfAbrupt(*nextValue*).
    h. Let *k* be the result of Get(*nextValue*, `"0"`).
    i. ReturnIfAbrupt(*k*).
    j. Let *v* be the result of Get(*nextValue*, `"1"`).
    k. ReturnIfAbrupt(*v*).
    l. Let *status* be the result of calling the [[Call]] internal method of *adder* with *map* as *thisArgument* and a List whose elements are *k* and *v* as *argumentsList*.
    m. ReturnIfAbrupt(*status*).

NOTE If the parameter *iterable* is present, it is expected to be an object that implements either an `entries` method or an @@iterator method that returns an iterator object that produces two element array-like objects 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.

15.15.1.2 new WeakMap (... argumentsList)

When `WeakMap` is called as part of a `new` expression it is a constructor: it initialises a newly created object.

`WeakMap` called as part of a new expression with argument list *argumentsList* performs the following steps:

Deleted: <#>15.15.1 Abstract Operations For WeakMap Objects¶
<#>15.15.1.1 WeakMapInitialisation¶
The abstract operation WeakMapInitialisation with arguments *obj* and *iterable* is used to initialize an objec as a map. It performs the following steps:¶
<#>If Type(*obj*) is not Object, throw a **TypeError** exception.¶
<#>If *obj* does not have already has a [[WeakMapData]] internal data property, throw a **TypeError** exception.¶
<#>If the result of calling the [[GetIsExtensible]] internal propertymethod of *obj* is **false**, throw a **TypeError** excepti
<#>If *iterable* is not **undefined**, then¶
<#>Let *iterable* be ToObject(*iterable*).¶
<#>ReturnIfAbrupt(*iterable*)¶
<#>Let *iterator* be the intrinsic symbol @@iterator.¶
<#>Let *itr* be the result of calling the Invoke(*iterable* o abstraction operation with *iterator*, *obj*, and an empty Li as arguments).¶
<#>ReturnIfAbrupt(*itr*).¶
<#>Let *adder* be the result of calling the [[Get]] interna method of(*obj* with argument, `"set"`).¶

Deleted: Called as a Function

Deleted: ize

Deleted: internal

Deleted:

Deleted: This premitspermits super invocation of the `WeakMap` constructor by `WeakMKap` subclasses. ¶

Deleted: *m*

Deleted: <#>If *m* is **undefined** or the intrinsic %WeakMapPrototype%¶

Deleted: Let *map* be the result of ToObject(*m*)

Deleted: ReturnIfAbrupt(*map*)

Deleted: n

Deleted: <#>Let *iterator* be the intrinsicwell-known symbol @@iterator.¶

Deleted: Invoke

Deleted: , `"next"`

Deleted: <#>If IteratorComplete(*next*) is **true**, then return NormalCompletion(*map*).¶

Deleted: <#>If Type(*next*) is not Object, then throw a **TypeError** exception.¶

Deleted: *next*

Deleted: *next*

Comment [AWB12224]: Note that using a method c for inserting pairs during initialization provides allows

Deleted: <#>Let *status* be the result of MapInitialisation with *map* and *iterable* as arguments.¶

Deleted: <#>Return *map*.¶

Deleted: <#>The WeakMap Constructor¶

Deleted: The `WeakMap` constructor is designed to be subclassable. It may be used as the value of an `exten`

Deleted: iterable = undefined

Deleted: args

Deleted: *args*

1. Let *F* be the WeakMap function object on which the **`new`** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **`new`** operator.
3. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If WeakMap is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.15.2 Properties of the WeakMap Constructor

The value of the [[Prototype]] internal data property of the WeakMap constructor is the Function prototype object (15.3.3).

Besides the **`length`** property (whose value is **0**), the WeakMap constructor has the following property:

15.15.2.1 WeakMap.prototype

The initial value of **`WeakMap.prototype`** is the WeakMap prototype object (15.15.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.15.2.2 WeakMap[@@create] ( )

The @@create method of a WeakMap object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, **`"%WeakMapPrototype%"`**, ([[WeakMapData]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.15.3 Properties of the WeakMap Prototype Object

The value of the [[Prototype]] internal data property of the WeakMap prototype object is the standard built-in Object prototype object (15.2.4). The WeakMap prototype object is an ordinary object. It does not have a [[WeakMapData]] internal data property.

15.15.3.1 WeakMap.prototype.constructor

The initial value of **`WeakMap.prototype.constructor`** is the built-in **`WeakMap`** constructor.

15.15.3.2 WeakMap.prototype.clear ()

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. If *M's* [[WeakMapData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Set the value of *M*'s [[WeakMapData]] internal data property to a new empty List.
6. Return **undefined**.

15.15.3.3 WeakMap.prototype.delete (key)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.

Deleted: be the be the

Deleted: with arguments

Deleted: and

Deleted: a

Deleted: <#>Let *map* be the result of the abstract operation ObjectCreate (15.2) with the intrinsic %WeakMapPrototype% as the argumen
<#>If *iterable* is not present, let *iterable* be **undefined**.¶
<#>Let *status* be the result of WeakMapInitialisation with *map* and *iterable* as arguments.¶
<#>ReturnIfAbrupt(*status*).¶
<#>Return *map*.¶

Deleted: NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that return an iterator object that produces two element array-like objects whose first element is a val that will be used as a WeakMap key and who second element is the value to associate wit that key.¶

Deleted: the internal properties and

Deleted: ,

Deleted: <#>Add a [[WeakMapData]] interna data property to *obj*.¶
<#>Set *obj*'s [[WeakMapData]] internal data property to a new empty List.¶

Comment [AWB14225]: Because the @@create method is essential to the integrity this "class" definition, just like the prototype property, it seems appripiate to freeze it in th same manner.

Deleted: false

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

Deleted: TypeError

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. If *entries* is **undefined**, then throw a **TypeError** exception.
6. If Type(*key*) is not Object, then throw a **TypeError** exception.
7. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
   a. If *p*.[[key]] and *key* are the same object, then
      i. Set *p*.[[key]] to empty.
      ii. Set *p*.[[value]] to empty.
      iii. Return **true**.
8. Return **false**.

NOTE The value **empty** is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

15.15.3.4 WeakMap.prototype.get (key)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. If *entries* is **undefined**, then throw a **TypeError** exception.
6. If Type(*key*) is not Object, then throw a **TypeError** exception.
7. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
   a. If *p*.[[key]] and *key* are the same object, then return *p*.[[value]]
8. Return **undefined**.

15.15.3.5 WeakMap.prototype.has (key)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. If *entries* is **undefined**, then throw a **TypeError** exception.
6. If Type(*key*) is not Object, then throw a **TypeError** exception.
7. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
   a. If *p*.[[key]] and *key* are the same object, then return **true**.
8. Return **false**.

15.15.3.6 WeakMap.prototype.set (key , value)

The following steps are taken:

1. Let *M* be the **this** value.
2. If Type(*M*) is not Object, then throw a **TypeError** exception.
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. If *entries* is **undefined**, then throw a **TypeError** exception.
6. If Type(*key*) is not Object, then throw a **TypeError** exception.
7. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
   a. If *p*.[[key]] and *key* are the same object, then
      i. Set *p*.[[value]] to *value*.
      ii. Return *M*.
8. Let *p* be the Record {[[key]]: *key*, [[value]]: *value*}
9. Append *p* as the last element of *entries*.
10. Return *M*.

Deleted: TypeError

Deleted: <#>Let *k* be ToObject(*key*).¶ <#>ReturnIfAbrupt(*k*).¶

Deleted: SameValue(

Deleted: ,

Deleted: *k*

Deleted:)

Deleted: result of calling ToObject with the

Deleted: the as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

Deleted: TypeError

Deleted: <#>Let *k* be ToObject(*key*).¶ <#>ReturnIfAbrupt(*k*).¶

Deleted: SameValue(

Deleted: ,

Deleted: *k*

Deleted:)

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

Deleted: TypeError

Deleted: <#>Let *k* be ToObject(*key*).¶ <#>ReturnIfAbrupt(*k*).¶

Deleted: SameValue(

Deleted: ,

Deleted:)

Deleted: result of calling ToObject with the

Deleted: as its argument

Deleted: TypeError

Deleted: ReturnIfAbrupt(*M*)

Deleted: TypeError

Deleted: <#>Let *k* be ToObject(*key*).¶ <#>ReturnIfAbrupt(*k*).¶

Deleted: SameValue(

Deleted: ,

Deleted:)

Deleted: **undefined**

Deleted: *k*

Deleted: **undefined**

15.15.3.7 WeakMap.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value **"WeakMap"**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.15.4 Properties of WeakMap Instances

WeakMap instances are ordinary objects that inherit properties from the WeakMap prototype. WeakMap instances also have a [[WeakMapData]] internal data property.

15.16 Set Objects

Set objects are collections of ECMAScript language values. A distinct value may only occur once as an element of a Set's collection. Distinct values are discriminated using a comparision algorithm that is selected when the Set is created.

A Set object can iterate its elements in insertion order. Set objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Set objects specification is only intended to describe the required observable semantics of Set objects. It is not intended to be a viable implementation model.

15.16.1 The Set Constructor

The Set constructor is the %Set% instrinsic object and the initial value of the **Set** property of the global object. When **Set** is called as a function rather than as a constructor, it initialises its **this** value with the internal state necessary to support the **Set.prototype** internal methods.

The **Set** constructor is designed to be subclassable. It may be used as the value in an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified **Set** behaviour must include a **super** call to **Set**.

15.16.1.1 Set (iterable = undefined, comparator = undefined)

When the **Set** function is called with optional arguments *iterable* and *comparator* the following steps are taken:

1. Let *set* be the **this** value.
2. If Type(*set*) is not Object then, throw a **TypeError** exception.
3. If *set* does not have a [[SetData]] internal data property, then throw a **TypeError** exception.
4. If *set's* [[SetData]] internal data property is not **undefined**, then throw a **TypeError** exception.
5. If *iterable* is not present, let *iterable* be **undefined**.
6. If *iterable* is **null**, then let *iter* be **undefined**.
7. Else,
   a. Let *iter* be the result of GetIterator(*iterable*).
   b. ReturnIfAbrupt(*iter*).
   c. Let *adder* be the result of Get(*set*, **"add"**).
   d. ReturnIfAbrupt(*adder*).
   e. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
8. If *comparator* is not present, let *comparator* be **undefined**.
9. If *comparator* is not **undefined**, then
   a. If *comparator* is not **"is"**, then throw a **RangeError** Exception.
10. Set *set's* [[SetData]] internal data property to a new empty List.
11. Set *set's* [[SetComparator]] internal data property to *comparator*.
12. If *iter* is **undefined**, then return *set*.
13. Repeat
    a. Let *next* be the result of IteratorNext(*iter*).
    b. ReturnIfAbrupt(*next*).

Deleted: .
Deleted: After initialisation by the WeakWeakMap constructor,
Deleted: of arbitrary
Deleted: arbitrary
Deleted: Each d
Deleted: , as determined using the SavmeValue algorithm,
Deleted: within
Deleted: s
Deleted: s
Deleted: the
Deleted:
Deleted: also
Comment [AWB7226]: TODO: need to define
Deleted: <#>15.16.1 Abstract Operatio For Set Objects¶
<#>15.16.1.1 SetInitialisation¶
The abstract operation SetInitialisation with arguments *obj* and *iterable* is used to initializ an object as a set instance. It performs the following steps:¶
<#>If Type(*obj*) is not Object, throw a **TypeEr** exception.¶
<#>If *obj* already hasdoes not have a [[SetData internal data property, throw a **TypeError** exception.¶
<#>If the result of calling the [[GetIsExtensible internal propertymethod of *obj* is **false**, throw a **TypeError** exception.¶
<#>If *iterable* is not **undefined**, then¶
<#>Let *iterable* be ToObject(*iterable*).¶
<#>ReturnIfAbrupt(*iterable*)¶
<#>Let *hasValues* be the result of HasProperty(calling the [[HasProperty]] inter method of *iterable* with argument, **"values**
<#>ReturnIfAbrupt(*hasValues*).¶
<#>If *hasValues* is **true**, then ¶
<#>Let *itr* be the result of calling the Invoke(*obj*, **abstraction operatic**
Deleted: Called as a Function
Deleted: ize
Deleted:
Deleted: This premitspermits super invocati of the **Set** constructor by **Set** subclasses.
Deleted: *O*
Deleted: <#>If *O* is **undefined** or the intrinsi
Deleted: <#>Let *set* be the result of
Deleted: neither **undefined** or
Deleted: <#>Let *iterator* be the intrinsicw
Deleted: Invoke
Deleted: , *iterator*
Deleted: Invoke
Deleted: , **"next"**

c. Let *done* be IteratorComplete(*next*).
d. ReturnIfAbrupt(*done*).
e. If *done* is **true**, then return *set*.
f. Let *nextValue* be IteratorValue(*next*).
g. ReturnIfAbrupt(*nextValue*).
h. Let *status* be the result of calling the [[Call]] internal method of *adder* with *set* as *thisArgument* and a List whose sole element is *nextValue* as *argumentsList*.
i. ReturnIfAbrupt(*status*).

15.16.1.2 new Set (... argumentsList)

When `Set` is called as part of a `new` expression it is a constructor: it initialises a newly created object.

`Set` called as part of a new expression with argument list *argumentsList* performs the following steps:

1. Let *F* be the Set function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If Set is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.16.2 Properties of the Set Constructor

The value of the [[Prototype]] internal data property of the Set constructor is the Function prototype object (15.3.3).

Besides the `length` property (whose value is **0**), the Set constructor has the following property:

15.16.2.1 Set.prototype

The initial value of `Set.prototype` is the intrinsic %SetPrototype% object (15.16.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.16.2.2 Set[@@create] ( )

The @@create method of a Set function object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%SetPrototype%"`, ([[SetData]] , [[SetComparator]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.16.3 Properties of the Set Prototype Object

The value of the [[Prototype]] internal data property of the Set prototype object is the standard built-in Object prototype object (15.2.4). The Set prototype object is an ordinary object. It does not have a [[SetData]] or a [[SetComparator]] internal data property.

15.16.3.1 Set.prototype.constructor

The initial value of `Set.prototype.constructor` is the built-in `Set` constructor.

Deleted: <#>If Type(*next*) is not Object, then throw a **TypeError** exception.¶

Deleted: IteratorComplete(*next*)

Deleted: NormalCompletion(

Deleted:)

Deleted: <#>ReturnIfAbrupt(*next*).¶

Comment [AWB12227]: Note that using a method c for inserting pairs during initialization allows subclasses be more expressive.

Deleted: whose

Deleted: *next*

Deleted: <#>Let *status* be the result of SetInitialisation with *set* and *iterable* as arguments.¶

Deleted: <#>Return *set*.¶

Deleted: NOTE If the parameter iterable is present, i expected to be an object that implements an @@iterat method that returns an iterator object that produces tw element array-like objects whose first element is a valu that will be used as an Map key and whose second element is the value to associate with that key.¶

Deleted: <#>The Set Constructor¶
When `Set` is called as part of a `new` expression it is a constructor: it initialises thea newly created object.¶

Deleted: The `Set` constructor is designed to be subclassable. It may be used as the value of an `exten` clause of a class declaration. Subclass constructors tha intended to inherit the specified `Set` behaviour must include a `super` call to `Set`.¶

Deleted: args

Deleted: iterable = undefined

Deleted: *args*

Deleted: <#>Let *set* be the result of the abstract operati ObjectCreate (15.2) with the intrinsic %SetPrototype% as argument.¶
<#>If *iterable* is not present, let *iterable* be **undefined**.¶
<#>Let *status* be the result of SetInitialisation with *set* an *iterable* as arguments.¶
<#>ReturnIfAbrupt(*status*).¶
<#>Return *set*.¶

Deleted: be the be the

Deleted: with arguments

Deleted: and

Deleted: a

Deleted: NOTE . If the parameter iterable is present, i expected to be an object that implements either a `values` method or an @@iterator method. Either method is expected to return an interator object that

Deleted: 15.3.4

Deleted: the internal properties and

Deleted: ,

Deleted: <#>Add a [[SetData]] internal data property to

Comment [AWB14228]: Because the @@create

Deleted: false

Deleted: s

ecma INTERNATIONAL

15.16.3.2 Set.prototype.add (value)

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. If *S's* [[SetComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each *e* that is an element of *entries*, in original insertion order
   a. If *e* is not empty and *same*(*e*, *value*) is **true**, then
      i. Return *S*.
9. Append *value* as the last element of *entries*.
10. Return *S*.

15.16.3.3 15.Set.prototype.clear ()

The following steps are taken:

1. Let *S* be **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. Repeat for each *e* that is an element of *entries*,
   a. Replace the element of *entries* whose value is *e* with an element whose value is empty.
7. Return **undefined**.

15.16.3.4 Set.prototype.delete (value)

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. If *S's* [[SetComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each *e* that is an element of *entries*, in original insertion order
   a. If *e* is not empty and *same*(*e*, *value*) is **true**, then
      i. Replace the element of *entries* whose value is *e* with an element whose value is empty.
      ii. Return **true**.
9. Return **false**.

NOTE The value **empty** is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

15.16.3.5 Set.prototype.entries ( )

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the CreateSetIterator abstract operation with arguments *S* and `"key+value"`.

Deleted: result of calling ToObject with the
Deleted: value as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*S*)
Deleted: TypeError
Deleted: <#>Repeat for each *p* that is an element of *entries*,¶
Deleted: *p*
Deleted: *S*
Deleted: Value
Deleted: *p*
Deleted: **undefined**
Deleted: *p*
Deleted: **undefined**
Deleted: the result of calling ToObject with t
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*S*)
Deleted: TypeError
Deleted: Record {[[key]], [[value]]} p
Deleted: <#>Set the value of *S*'s [[SetData]] internal data data property to a new empty List.¶
Deleted: Map
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*S*)
Deleted: TypeError
Deleted: *S*
Deleted: Value
Deleted: TypeError

NOTE For iteration purposes, a Set appears similar to a Map where each entry has the same value for its key and value.

15.16.3.6 Set.prototype.forEach (callbackfn , thisArg = undefined)

callbackfn should be a function that accepts three arguments. **forEach** calls *callbackfn* once for each value present in the set object, in value insertion order. *callbackfn* is called only for values of the Set which actually exist; it is not called for keys that have been deleted from the set.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

NOTE 1 If *callbackfn* is an Arrow Function, **this** was lexically bound when the function was created so *thisArg* will have no effect.

callbackfn is called with three arguments: the first two arguments are a value contained in the Set. The same value of passed for both arguments. The Set object being traversed is passed as the third argument.

NOTE 2 The *callbackfn* is called with three arguments to be consistent with the call back functions used by **forEach** methods for Map and Array. For Sets, each item value is considered to be both the key and the value.

forEach does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

NOTE 3 Each value is normally visited only once. However, a value will be revisited if it is deleted after it has been visited and then re-added before the to **forEach** call completes. Values that are deleted after the call to **forEach** begins and before being visited are not visited unless the value is added again before the to **forEach** call completes. New values added, after the call to **forEach** begins are visited.

When the **forEach** method is called with one or two arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
6. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
7. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
8. Repeat for each *e* that is an element of *entries*, in original insertion order
    a. If *e* is not empty, then
        i. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *e*, *e*, and *S* as *argumentsList*.
        ii. ReturnIfAbrupt(*funcResult*).
9. Return **undefined**.

The **length** property of the **forEach** method is **1**.

15.16.3.7 Set.prototype.has (value)

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. If *S's* [[SetComparator]] internal data property is **undefined**, then let *same* be the abstract operation SameValueZero.
7. Else, let *same* be the abstract operation SameValue.
8. Repeat for each *e* that is an element of *entries*,

Deleted: two
Deleted: map
Deleted: two
Deleted: value of the item
Deleted: and t
Deleted: if
Deleted: result of calling ToObject with the
Deleted: value as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*S*)
Deleted: TypeError
Deleted: result of calling ToObject with the
Deleted: as its argument
Deleted: TypeError
Deleted: ReturnIfAbrupt(*S*)
Deleted: TypeError

a. If *e* is not empty and *same*(*e*, *value*), then return **true**.

9. Return **false**.

15.16.3.8 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.

15.16.3.9 get Set.prototype.size

Set.prototype.size is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. Let *count* be 0.
7. For each *e* that is an element of *entries*
   a. If *e* is not empty then
      i. Set *count* to *count*+1.
8. Return *count*.

15.16.3.10 Set.prototype.values ( )

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the CreateSetIterator abstract operation with argument *S* and **"value"**.

15.16.3.11 Set.prototype [@@iterator] ( )

The initial value of the @@iterator property is the same function object as the initial value of the **values** property.

15.16.3.12 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** }.

15.16.4 Properties of Set Instances

Set instances are ordinary objects that inherit properties from the Set prototype. After initialisation by the Set constructor, Set instances also have a [[SetData]] internal data property and a [[SetComparator]] internal data property.

15.16.5 Set Iterator Object Structure

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.

15.16.5.1 CreateSetIterator Abstract Operation

Several methods of Set objects return Iterator objects. The abstract operation CreateSetIterator with arguments *set* and *kind* is used to create such iterator objects. It performs the following steps:

1. Let *S* be the result of calling ToObject(*set*).
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[SetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
6. Let *iterator* be the result of ObjectCreate(%SetIteratorPrototype%, ([[IteratedSet]], [[SetNextIndex]], [[SetIterationKind]])).
7. Set *iterator's* [[IteratedSet]] internal data property to *S*.
8. Set *iterator's* [[SetNextIndex]] internal data property to 0.
9. Set *iterator's* [[SetIterationKind]] internal data property to *kind*.
10. Return *iterator*.

15.16.5.2 The Set Iterator Prototype

All Set Iterator Objects inherit properties from a common Set Iterator Prototype object. The [[Prototype]] internal data property of the Set Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Set Iterator Prototype as the following properties:

15.16.5.2.1 *SetIterator*.prototype.constructor

15.16.5.2.2 *SetIterator*.prototype.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 properties of a Set Iterator Instance (15.16.7.1.2), throw a **TypeError** exception.
4. Let *s* be the value of the [[IteratedSet]] internal data property of *O*.
5. Let *index* be the value of the [[SetNextIndex]] internal data property of *O*.
6. Let *itemKind* be the value of the [[SetIterationKind]] internal data property of *O*.
7. Assert: *s* has a [[SetData]] internal data property and *s* has been initialised so the value of [[SetData]] is not **undefined**.
8. Let *entries* be the List that is the value of the [[SetData]] internal data property of *s*.
9. Repeat while *index* is less than the total number of element of *entries*. The number of elements must be redetermined each time this method is evaluated.
    a. Let *e* be the element at 0-origined insertion position *index* of *entries*.
    b. Set *index* to *index*+1;
    c. Set the [[SetNextIndex]] internal data property of *O* to *index*.
    d. If *e* is not empty, then
        i. If *itemKind* is `"key+value"` then,
            1. Let *result* be the result of the abstract operation ArrayCreate with argument 2.
            2. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
            3. Call CreateOwnDataProperty(*result*, `"0"`, *e*) .
            4. Call CreateOwnDataProperty(*result*, `"1"`, *e*).
            5. Return CreateItrResultObject(*result*, **false**).
        ii. Return *e*.
10. Return CreateItrResultObject(**undefined**, **true**).

15.16.5.2.3 *SetIterator*.prototype.@@iterator ( )

The following steps are taken:

1. Return the **this** value.

Deleted: The **value** and @@iterator
Deleted: interator
Deleted: and
Deleted: TypeError
Deleted: *itr*
Deleted: the abstract operation
Deleted: with the intrinsic object
Deleted: as its argument
Deleted:]]
Deleted: *itr*
Deleted: Add a
Deleted: *itr* with value
Deleted: *itr*
Deleted: Add a
Deleted: rn
Deleted: to *itr* with value
Deleted: *itr*
Deleted: *itr*
Comment [AWB10230]: TODO: need to decide wh to use for a constructor for these sort of objects.
Deleted:
Deleted: z
Deleted: Map
Deleted: If *itemKind* is `"key"` then, r
Deleted: Completion {[[type]]: throw, [[value]]: %StopIteration%, [[target]]: empty}

15.16.5.2.4 ***SetIterator*.prototype.@@toStringTag**

The initial value of the @@toStringTag property is the string value **"Set Iterator"**.

15.16.5.3 **Properties of Set Iterator Instances**

Set Iterator instances inherit properties from the Set Iterator prototype (the intrinsic, %SetIteratorPrototype%.) Set Iterator instances are initially created with the internal properties specified in Table 38.

Table 38 — Internal Data Properties of Set Iterator Instances

Internal Data Property Name	Description
[[IteratedSet]]	The Set object that is being iterated.
[[SetNextIndex]]	The integer index of the next Set data element to be examined by this iteration.
[[SetIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: **"key"**, **"value"**, **"key+value"**. **"key"** and **"value"** have the same meaning.

15.17 WeakSet Objects

WeakSet objects are collections of ECMAScript objects. A distinct object may only occur once as an element of a WeakSet's collection. A WeakSet may be queried to see if it contains a specific object, but no mechanisms is provided for enumerating the objects it holds. If an object that is contain by a WeakSet is only reachable by following a chain of references that start within that WeakMap, 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 objcet.

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 15.15.

15.17.1 The WeakSet Constructor

The WeakSet constructor is the %WeakSet% instrinsic object and the initial value of the **WeakSet** property of the global object. When **WeakSet** is called as a function rather than as a constructor, it initialises its **this** value with the internal state necessary to support the **WeakSet.prototype** internal methods.

The **WeakSet** constructor is designed to be subclassable. It may be used as the value in an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified **WeakSet** behaviour must include a **super** call to **WeakSet**.

15.17.1.1 **WeakSet (iterable = undefined)**

When the **WeakSet** function is called with optional argument *iterable* the following steps are taken:

1. Let *set* be the **this** value.

2. If Type(*set*) is not Object then, throw a **TypeError** exception.
3. If *set* does not have a [[WeakSetData]] internal data property, then throw a **TypeError** exception.
4. If *set's* [[WeakSetData]] internal data property is not **undefined**, then throw a **TypeError** exception.
5. If *iterable* is not present, let *iterable* be **undefined**.
6. If *iterable* is **null**, then let *iter* be **undefined**.
7. Else,
    a. Let *iter* be the result of GetIterator(*iterable*).
    b. ReturnIfAbrupt(*iter*).
    c. Let *adder* be the result of Get(*set*, **"add"**).
    d. ReturnIfAbrupt(*adder*).
    e. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
8. Set *set's* [[WeakSetData]] internal data property to a new empty List.
9. If *iter* is **undefined**, then return *set*.
10. Repeat
    a. Let *next* be the result of IteratorNext(*iter*).
    b. ReturnIfAbrupt(*next*).
    c. Let *done* be IteratorComplete(*next*).
    d. ReturnIfAbrupt(*done*).
    e. If *done* is **true**, then return NormalCompletion(*set*).
    f. Let *nextValue* be IteratorValue(*next*).
    g. ReturnIfAbrupt(*nextValue*).
    h. Let *status* be the result of calling the [[Call]] internal method of *adder* with *set* as *thisArgument* and a List whose sole element is *nextValue* as *argumentsList*.
    i. ReturnIfAbrupt(*status*).

15.17.1.2 new WeakSet (... argumentsList)

When **WeakSet** is called as part of a **new** expression it is a constructor: it initialises a newly created object.

WeakSet called as part of a new expression with argument list *argumentsList* performs the following steps:

1. Let *F* be the **WeakSet** function object on which the **new** operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the **new** operator.
3. Return the result of OrdinaryConstruct(*F*, *argumentsList*).

If WeakSet is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.17.2 Properties of the WeakSet Constructor

The value of the [[Prototype]] internal data property of the WeakSet constructor is the Function prototype object (15.3.3).

Besides the **length** property (whose value is **0**), the WeakSet constructor has the following property:

15.17.2.1 WeakSet.prototype

The initial value of **WeakSet.prototype** is the intrinsic %WeakSetPrototype% object (15.17.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.17.2.2 WeakSet [@@create] ( )

The @@create method of a WeakSet function object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, **"%WeakSetPrototype%"**, ([[WeakSetData]])).

3. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.17.3 Properties of the WeakSet Prototype Object

The value of the [[Prototype]] internal data property of the WeakSet prototype object is the standard built-in Object prototype object (15.2.4). The WeakSet prototype object is an ordinary object. It does not have a [[WeakSetData]] internal data property.

15.17.3.1 WeakSet.prototype.constructor

The initial value of `WeakSet.prototype.constructor` is the built-in `Set` constructor.

15.17.3.2 WeakSet.prototype.add (value)

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[WeakSetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[WeakSetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[WeakSetData]] internal data property.
6. Repeat for each *e* that is an element of *entries*, in original insertion order
    a. If *e* is not empty and SameValue(*e*, *value*) is **true**, then
        i. Return *S*.
7. Append *value* as the last element of *entries*.
8. Return *S*.

15.17.3.3 WeakSet.prototype.clear ()

The following steps are taken:

1. Let *S* be **this** value.
2. If Type(*S*) is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[WeakSetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[WeakSetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[WeakSetData]] internal data property.
6. Repeat for each e that is an element of *entries*,
    a. Replace the element of *entries* whose value is *e* with an element whose value is empty.
7. Return **undefined**.

15.17.3.4 WeakSet.prototype.delete (value)

The following steps are taken:

10. Let *S* be the **this** value.
11. If Type(*S*) is not Object, then throw a **TypeError** exception.
12. If *S* does not have a [[WeakSetData]] internal data property throw a **TypeError** exception.
13. If *S's* [[
14. WeakSetData]] internal data property is **undefined**, then throw a **TypeError** exception.
15. Let *entries* be the List that is the value of *S*'s [[WeakSetData]] internal data property.
16. Repeat for each *e* that is an element of *entries*, in original insertion order
    a. If *e* is not empty and SameValue(*e*, *value*) is **true**, then
        i. Replace the element of *entries* whose value is *e* with an element whose value is empty.
        ii. Return **true**.
17. 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.

15.17.3.5 WeakSet.prototype.has (value)

The following steps are taken:

1. Let *S* be the **this** value.
2. If Type(*S)* is not Object, then throw a **TypeError** exception.
3. If *S* does not have a [[WeakSetData]] internal data property throw a **TypeError** exception.
4. If *S's* [[WeakSetData]] internal data property is **undefined**, then throw a **TypeError** exception.
5. Let *entries* be the List that is the value of *S*'s [[WeakSetData]] internal data property.
6. Repeat for each *e* that is an element of *entries*,
   a. If *e* is not empty and SameValue(*e*, *value*), then return **true**.
7. Return **false**.

15.17.3.6 WeakSet.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value **"WeakSet"**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.17.4 Properties of WeakSet Instances

Set instances are ordinary objects that inherit properties from the WeakSet prototype. After initialisation by the WeakSet constructor, WeakSet instances also have a [[WeakSetData]] internal data property.

15.18 The Reflect Module

This is a place holder for the material in http://wiki.ecmascript.org/doku.php?id=harmony:reflect_api

Comment [AWB16231]: TODO

Comment [AWB10232]: TODO

15.18.1 Exported Function Properties Reflecting the Essentional Internal Methods

15.18.1.1 Reflect.getPrototypeOf (target)

When the `getPrototypeOf` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[GetInheritance]] internal method of *obj*.

Deleted: e

15.18.1.2 Reflect.setPrototypeOf (target, proto)

When the `setPrototypeOf` function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. If Type(*proto*) is not Object and *proto* is not **null**, then throw a **TypeError** exception
4. Return the result of calling the [[SetInheritance]] internal method of *obj* with argument *proto*.

15.18.1.3 Reflect.isExtensible (target)

When the `isExtensible` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsExtensible]] internal method of *obj*.

Deleted: the [[

Deleted: IsExtensible]] internal method of(*obj*)

15.18.1.4 Reflect.preventExtensions (target)

When the `preventExtensions` function is called with argument *target*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[PreventExtensions]] internal method of *obj*.

15.18.1.5 Reflect.has (target, propertyKey)

When the `hasOwn` function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of calling the [[HasProperty]] internal method of *obj* with argument *key*.

15.18.1.6 Reflect.hasOwn (target, propertyKey)

When the `hasOwn` function is called with arguments *target* and *propertyKey*, the following steps are taken:

6. Let *obj* be ToObject(*target*).
7. ReturnIfAbrupt(*obj*).
8. Let *key* be ToPropertyKey(*propertyKey*).
9. ReturnIfAbrupt(*key*).
10. Return the result of calling the [[HasOwnProperty]] internal method of *obj* with argument *key*.

15.18.1.7 Reflect.getOwnPropertyDescriptor(target, propertyKey)

When the `getOwnPropertyDescriptor` function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *key*.
6. ReturnIfAbrupt(*desc*).
7. Return the result of calling FromPropertyDescriptor(*desc*).

15.18.1.8 Reflect.get (target, propertyKey, receiver=target)

When the `get` function is called with arguments *target*, *propertyKey*, and *receiver* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. If *receiver* is not present, then
   a. Let *receiver* be *target*.
6. Return the result of calling the [[Get]] internal method of *obj* with arguments *key*, and *receiver*.

Deleted: P

15.18.1.9 Reflect.set (target, propertyKey, V, receiver=target)

When the `set` function is called with arguments *target*, *V*, *propertyKey*, and *receiver* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. If *receiver* is not present, then

a. Let *receiver* be *target*.

6. Return the result of calling the [[Set]] internal method of *obj* with arguments *key*, *V*, and *receiver*.

15.18.1.10 Reflect.invoke (target, propertyKey, argumentsList, receiver=target)

When the `invoke` function is called with arguments *target*, *propertyKey*, *argumentsList*, and *receiver* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. If *receiver* is not present, then
   a. Let *receiver* be *target*.
6. Let *argList* be the result of CreateListFromArrayLike(*argumentsList*).
7. ReturnIfAbrupt(*argList*).
8. Return the result of calling the [[Invoke]] internal method of *obj* with arguments *key*, *argList*, and *receiver*.

15.18.1.11 Reflect.deleteProperty (target, propertyKey)

When the `deleteProperty` function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of calling the [[Delete]] internal method of *obj* with argument *key*.

15.18.1.12 Reflect.defineProperty(target, propertyKey, attributes)

When the `defineProperty` function is called with arguments *target*, *propertyKey*, and *attributes* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Let *desc* be the result of calling ToPropertyDescriptor with *attributes* as the argument.
6. ReturnIfAbrupt(*desc*).
7. Return the result of calling the [[DefineOwnProperty]] internal method of *obj* with arguments *key*, and *desc*.

15.18.1.13 Reflect.enumerate (target)

When the `enumerate` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *iterator* be the result of calling the [[Enumerate]] internal method of *obj*.
4. Return *iterator*.

15.18.1.14 Reflect.ownKeys (target)

When the `ownKeys` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.

Deleted: P

Deleted: A

Deleted: *A*

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

Deleted: *A*

Deleted: *itr*

Deleted: *itr*

Deleted: <#>15.17.1.12 Reflect.keys (target)¶
When the `keys` function is called with argument *target* the following steps are taken:¶
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Let *keys* be the result of calling the [[Keys]] internal method of *obj*.¶
<#>ReturnIfAbrupt(*keys*).¶
<#>Return CreateArrayFromList(*keys*).¶

Deleted: getOwnPropertyNames

Deleted: `getO`

Deleted: `PropertyNames`

Deleted: <#>Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.¶
<#>ReturnIfAbrupt(*keys*).¶

Deleted: Return CreateArrayFromList(*keys*).

15.18.2 Proxy Objects

15.19 The "std:iteration" Module

The **"std:iteration"** module defines built-in objects that support the for-of statement and similar iteration use cases. It also defines the built-in objects that support Generator Functions.

15.19.1 Common Iteration Interfaces

An interface is a set of object property keys whose associated values match a specific specification. Any object that provides all the properties of an interface in conformance to the interface's specification *conforms* to that interface. An interface isn't represented by a single object and there may be many distinctly implemented objects that conform to any interface. An individual object may conform to multiple interfaces.

15.19.1.1 The *Iterable* Iterface

The *Iterable* interface includes the following property:

Property	Value	Requirements
`@@iterator`	A zero arguments function that returns an object.	The function returns an object that conforms to the *iterator* interface.

15.19.1.2 The *Iterator* Interface

The *Iterator* interface includes the following properties:

Property	Value	Requirements
`next`	A function that returns an object.	The function returns an object that conforms to the *ItrResult* interface.

NOTE Arguments may be passed to the next function but their interpretation and validity is dependent upon the target Iterator. Generic use of Iterators should not pass any arguments.

15.19.1.3 The *ItrResult* Iterface

The *ItrResult* interface includes the following properties:

Property	Value	Requirements
`done`	Either **true** or **false**.	This is the result status of the 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.
`value`	Any ECMAScript languge value.	If done is **false**, this is the current iteration element value. If done is **true**, this is the return value of the iterator, if it supplied one. If the iterator does not have a return value, `value` is **undefined**. In that case, the `value` property may be absent form the conforming object if it does not inherit an explicit `value` property.

Deleted: <#>15.17.1.14 Reflect.freeze (target)¶
When the `freeze` function is called with argument *target* the following steps are taken
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Return the result of calling the [[Freeze]] internal method of *obj*.¶
<#>15.17.1.15 Reflect.seal (target)¶
When the `seal` function is called with argument *target* the following steps are taken
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Return the result of calling the [[FreezeSea internal method of *obj*.¶
<#>15.17.1.16 Reflect.isFrozen (targe
When the `isFrozen` function is called with argument *target* the following steps are taken
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Return the result of calling the [[IsFrozen]] internal method of *obj*.¶
<#>15.17.1.17 Reflect.isSealed (targe
When the `isSealed` function is called with argument *target* the following steps are taken
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Return the result of calling the [[IsSealed]] internal method of *obj*.¶
15.18

15.19.2 "std:iteration" Exports

The **`"std:iteration"`** module exports the names:

- iterator
- GeneratorFunction
- Generator

15.19.3 GeneratorFunction Objects

`Generator Function` objects are constructor functions that are usually created by evaluating *GeneratorDeclaration*, *GeneratorExpression*, and *GeneratorMethod* syntactic productions. They may also be created by calling the **`GeneratorFunction`** constructor.

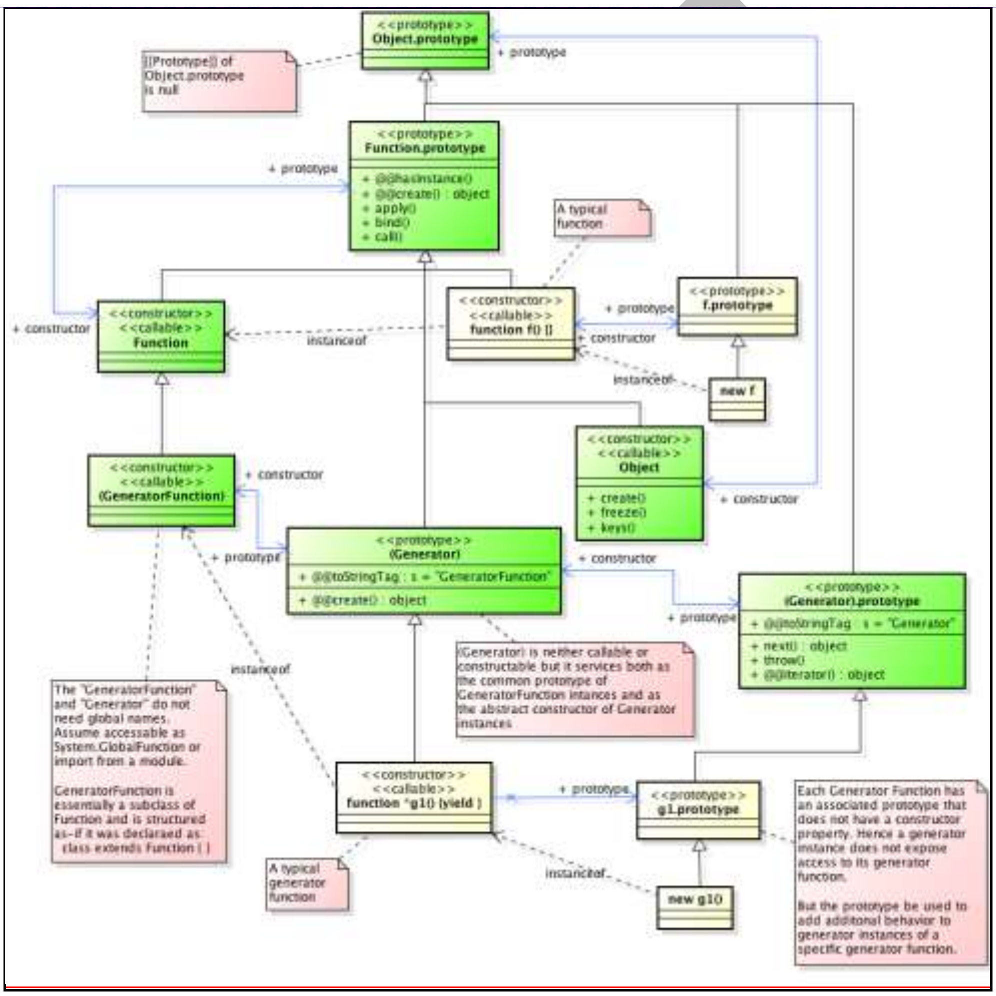

Figure 2 (Informative) -- Generator Objects Relationships

Deleted: I

Deleted: $

Deleted: <#>15.17.2.1 Reflect.has (target, propertyKey)¶
When the **`has`** function is called with arguments *target* and *propertyKey*, the following steps are taken:¶
<#>Let *obj* be ToObject(*target*).¶
<#>ReturnIfAbrupt(*obj*).¶
<#>Let *key* be ToPropertyKey(*propertyKey*).¶
<#>ReturnIfAbrupt(*key*).¶
<#>Return the result of HasProperty(*obj*, *key*).¶
<#>15.17.2.1 Reflect.instanceOf (target, O)¶
When the **`instanceOf`** function is called with argumen *target* and *O*, the following steps are taken:¶
<#>Return the result of OrdinaryInstanceOf(*target*, *O*).¶

Deleted: ¶

Comment [AWB15233]: Before final publication we should try to get a vector graphics version of this diagra

15.19.3.1 The GeneratorFunction Constructor

The GeneratorFunction constructor is the %GeneratorFunction% instrinsic object and the value of the name `GeneratorFunction` exported from the built-in module `"std:iteration"`. When `GeneratorFunction` is called as a function rather than as a constructor, it creates and initialises a new GeneratorFunction object. Thus the function call `GeneratorFunction (…)` is equivalent to the object creation expression `new GeneratorFunction (…)` with the same arguments. However, if the **this** value value passed in the call is an Object with an [[Code]] internal data property whose value is **undefined**, it initialises the **this** value using the argument values. This permits `GeneratorFunction` to be used both as factory method and to perform constructor instance initialization.

Deleted: ize

`GeneratorFunction` may be subclassed and subclass constructors may perform a `super` invocation of the `GeneratorFunction` constructor to initialise subclass instances. However, all syntactic forms for defining generator function objects create direct instances of `GeneratorFunction`. There is no syntactic means to create instances of `GeneratorFunction` subclasses.

Deleted: ize

15.19.3.1.1 GeneratorFunction (p1, p2, … , pn, body)

The last argument specifies the body (executable code) of a generator function; any preceding arguments specify formal parameters.

When the `GeneratorFunction` function is called with some arguments *p1*, *p2*, … , *pn*, *body* (where *n* might be 0, that is, there are no "*p*" arguments, and where *body* might also not be provided), the following steps are taken:

1. Let *argCount* be the total number of arguments passed to this function invocation.
2. Let *P* be the empty String.
3. If *argCount* = 0, let *bodyText* be the empty String.
4. Else if *argCount* = 1, let *bodyText* be that argument.
5. Else *argCount* > 1,
    a. Let *firstArg* be the first argument.
    b. Let *P* be ToString(*firstArg*).
    c. ReturnIfAbrupt(*P*).
    d. Let *k* be 2.
    e. Repeat, while *k* < *argCount*
        i. Let *nextArg* be the *k*'th argument.
        ii. Let *nextArgString* be ToString(*nextArg*).
        iii. ReturnIfAbrupt(*nextArgString*).
        iv. Let *P* be the result of concatenating the previous value of *P*, the String `","` (a comma), and *nextArgString*.
        v. Increase *k* by 1.
    f. Let *bodyText* be the *k*'th argument.
6. Let *bodyText* be ToString(*bodyText*).
7. ReturnIfAbrupt(*bodyText*).
8. Let *parameters* be the result of parsing *P*, interpreted as UTF-16 encoded Unicode text as described in 8.4, using *FormalParameters* as the goal symbol. Throw a **SyntaxError** exception if the parse fails.
9. Let *funcBody* be the result of parsing *bodyText*, interpreted as UTF-16 encoded Unicode text as described in 8.4, using *FunctionBody* as the goal symbol. Throw a **SyntaxError** exception if the parse fails or if any static semantics errors are detected.
10. If *funcBody* Contains *YieldExpression* is **false**, then throw a **SyntaxError** exception.
11. If IsSimpleParameterList of *parameters* is **false** and any element of the BoundNames of *parameters* also occurs in the VarDeclaredNames of *funcBody*, then throw a **SyntaxError** exception.
12. If any element of the BoundNames of *parameters* also occurs in the LexicallyDeclaredNames of *funcBody*, then throw a **SyntaxError** exception.
13. Let *scope* be the Global Environment.
14. Let *F* be the **this** value.
15. If Type(*F*) is not Object or if *F* does not have a [[Code]] internal data property or if the value of [[Code]] is not **undefined**, then
    a. Let *F* be the result of calling FunctionAllocate with arguments `%Generator%` and `"generator"`.

16. If the value of *F's* [[FunctionKind]] internal data property is not `"generator"`, then throw a **TypeError** exception.
17. Using *funcBody* as the *FunctionBody* production, let *body* be the supplemental syntactic grammar production: *GeneratorBody* **:** *FunctionBody*.
18. Perform the FunctionInitialise abstract operation with arguments *F*, Normal, *parameters*, *body*, *scope*, and *strict*.
19. Let *prototype* be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
20. Perform the abstract operation MakeConstructor with arguments *F*, **true**, and *prototype*.
21. Return *F*.

Deleted: ize

A `prototype` property is automatically created for every function created using the `GeneratorFunction` constructor, to provide for the possibility that the function will be used as a constructor.

15.19.3.1.2 new GeneratorFunction (... argumentsList)

When `GeneratorFunction` is called as part of a `new` expression, it creates and initialises a newly created object.

1. Let *F* be the `GeneratorFunction` function object on which the `new` operator was applied.
2. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (*F*, *argumentsList*).

If `GeneratorFunction` is implemented as an ordinary function object, its [[Construct]] internal method will perform the above steps.

15.19.3.2 Properties of the GeneratorFunction Constructor

The GeneratorFunction constructor is a built-in Function object that inherits from the Function constructor. The value of the [[Prototype]] internal data property of the GeneratorFunction constructor is the intrinsic object %Function%.

The value of the [[Extensible]] internal data property of the GeneratorFunction constructor is **true**.

The GeneratorFunction constructor has the following properties:

15.19.3.2.1 GeneratorFunction.prototype

The initial value of `GeneratorFunction.prototype` is %Generator%, the standard built-in GeneratorFunction prototype.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.19.3.2.2 GeneratorFunction.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.19.3.2.3 GeneratorFunction[@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *proto* be the result of GetPrototypeFromConstructor(*F*, `"%Generator%"`).
3. ReturnIfAbrupt(*proto*).
4. Let *obj* be the result of calling FunctionAllocate with agument *proto* and `"generator"`.
5. Return *obj*.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE The GeneratorFunction `@@create` function is intentionally generic; it does not require that its **this** value be the GeneratorFunction constructor object. It can be transferred to other constructor functions for use as a `@@create` method. When used with other constructors, this function will create a function object whose [[Prototype]] value is obtained from the associated constructor.

15.19.3.3 Properties of the GeneratorFunction Prototype Object

The GeneratorFunction prototype object is an ordinary object. It is not a function object and does not have a [[Code]] internal data property or any other of the internal data properties listed in Table 13. In addition to being the value of the prototype property of the %GeneratorFunction% intrinsic and is itself the %Generator% instrinsic.

The value of the [[Prototype]] internal data property of the GeneratorFunction prototype object is the %FunctionPrototype% instrinsic object. The initial value of the [[Extensible]] internal data property of the GeneratorFunction prototype object is **true**.

15.19.3.3.1 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** }.

15.19.3.3.2 GeneratorFunction.prototype.prototype

The value of `GeneratorFunction.prototype.prototype` is the %GeneratorPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.19.3.3.3 GeneratorFunction.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value **"GeneratorFunction"**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.19.3.3.4 GeneratorFunction.prototype [@@create] ( )

The @@create method of an object *F* performs the following steps:

1. Let *F* be the **this** value.
2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, `"%GeneratorPrototype%"`, ([[GeneratorState]], [[GeneratorContext]])).
3. Return *obj*.

This property has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.19.3.4 GeneratorFunction Instances

Every GeneratorFunction instance is an ordinary function object and has the internal data properties listed in Table 13. The value of the [[FunctionKind]] internal data property for all such instances is "`generator`".

The GeneratorFunction instances have the following own properties:

15.19.3.4.1 length

The value of the `length` property is an integer that indicates the typical number of arguments expected by the GeneratorFunction. However, the language permits the function to be invoked with some other number of

Deleted:

arguments. The behaviour of a GeneratorFunction when invoked on a number of arguments other than the number specified by its `length` property depends on the function.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

15.19.3.4.2 **prototype**

Whenever a GeneratorFunction instance is created another ordinary object is also created and is the initial value of the generator function's `prototype` property. The value of the prototype property is used to initialise the [[Prototype]] internal data property of a newly created Generator object before the generator function object is invoked as a constructor for that newly created object.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

NOTE Unlike function instances, the object that is the value of the a GeneratorFunction's `prototype` property does not have a `constructor` property whose value is the GeneratorFunction instance.

15.19.4 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%.

15.19.4.1 **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 GeneratorFrunction.prototype).

The Generator prototype is an ordinary object. It is not a Generator instance and does not have a [[GeneratorState]] internal data property.

The value of the [[Prototype]] internal data property of the Generator prototype object is the intrinsic object %ObjectPrototype% (15.2.4). The initial value of the [[Extensible]] internal data property of the Function prototype object is **true**.

All Generator instances indirectly inherit properties of the Generator prototype object.

15.19.4.1.1 **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** }.

15.19.4.1.2 **Generator.prototype.next (value)**

The `next` method performs the following steps:

1. Let *g* be the **this** value.
2. Return the result of GeneratorResume(*g*, *value*).

15.19.4.1.3 **Generator.prototype.throw (exception)**

The `throw` method performs the following steps:

1. Let *g* be the **this** value.

2. If Type(*generator*) is not Object, then throw a **TypeError** exception.
3. If *generator* does not have a [[GeneratorState]] internal data property, then throw a **TypeError** exception.
4. Let *state* be the value of *generator's* [[GeneratorState]] internal data property.
5. Assert: *generator* also has a [[GeneratorContext]] internal data property.
6. If *state* is neither `"suspendedStart"` or `"suspendedYield"`, then throw a **TypeError** exception.
7. Let *E* be Completion {[[type]]: throw, [[value]]: *exception*, [[target]]: empty}.
8. If *state* is `"suspendedStart"` then,
    a. Set *generator's* [[GeneratorState]] internal data property to `"completed"`.
    b. Once a generator enters the `"completed"` state it never leaves it and its associated execution context is never resumed. Any execution state associated with *generator* can be discard at this point.
    c. Return *E*.
9. Let *genContext* be value of *generator's* [[GeneratorContext]] internal data property.
10. Let *methodContext* be the running execution context.
11. Suspend *methodContext*.
12. Set *generator's* [[GeneratorState]] internal data property to `"executing"`.
13. Push *genContext* onto the execution context stack; *genContext* is now the running execution context.
14. Resume the suspended evaluation of *genContext* using *E* as the result of the operation that suspended it. Let *result* be the value returned by the resumed compation.
15. Assert: When we return here, *genContext* has already been removed from the execution context stack and *methodContext* is the currently running execution context.
16. Return *result*.

15.19.4.1.4 Generator.prototype [@@iterator] ( )

The following steps are taken:

1. Return the **this** value.

15.19.4.1.5 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** }.

15.19.4.2 Properties of Generator Instances

Generator instances are initially created with the internal data properties described in Table 37.

Table 39 — Internal Data Properties of Generator Instances

Internal Data Property Name	Description
[[GeneratorState]]	The current execution state of the generator. The possible values are: **undefined**, `"suspendedStart"`, `"suspendedYield"`, `"executing"`, and `"completed"`.
[[GeneratorContext]]	The execution context that is used when executing the code of this generator.

15.19.4.3 Iteration Related Abstract Operations

15.19.4.3.1 GeneratorStart (generator, generatorBody)

1. Assert: The value of *generator's* [[GeneratorState]] internal data property 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:

Deleted: 15.19.4.2.1 @@iterator ¶
The value of the @@iterator own property of Generator instance is that Generator instance
This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

Deleted: <#>15.19.4.3 Iteration Related Abstract Operations¶
15.19.4.3.1

a. Let *result* be the result of evaluating *generatorBody*.
b. Assert: If we return here, the generator either threw an exception or performed either an implicit or explicit return.
c. Remove *genContext* from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.
d. Set *generator's* [[GeneratorState]] internal data property to `"completed"`.
e. Once a generator enters the `"completed"` state it never leaves it and its associated execution context is never resumed. Any execution state associated with *generator* can be discard at this point.
f. ReturnIfAbrupt(*result*).
g. Return CreateItrResultObject(*result*, **true**).

5. Set *generator's* [[GeneratorContext]] internal data property to *genContext*.
6. Set *generator's* [[GeneratorState]] internal data property to `"suspendedStart"`.
7. Return NormalCompletion(*generator*).

Deleted: *g*

15.19.4.3.2 GeneratorResume (generator, value)

The abstract operation GeneratorResume with arguments *generator* and *value* performs the following steps:

1. If Type(*generator*) is not Object, then throw a **TypeError** exception.
2. If *generator* does not have a [[GeneratorState]] internal data property, then throw a **TypeError** exception.
3. Let *state* be the value of *generator's* [[GeneratorState]] internal data property.
4. Assert: *generator* also has a [[GeneratorContext]] internal data property.
5. If *state* is neither `"suspendedStart"` or `"suspendedYield"`, then throw a **TypeError** exception.
6. If *state* is `"suspendedStart"` and *value* is not **undefined**, then throw a **TypeError** exception.
7. Let *genContext* be value of *generator's* [[GeneratorContext]] internal data property.
8. Let *methodContext* be the running execution context.
9. Suspend *methodContext*.
10. Set *generator's* [[GeneratorState]] internal data property to `"executing"`.
11. Push *genContext* onto the execution context stack; *genContext* is now the running execution context.
12. 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.
13. Assert: When we return here, *genContext* has already been removed from the execution context stack and *methodContext* is the currently running execution context.
14. Return *result*.

15.19.4.3.3 GeneratorYield (itrNextObj)

The abstract operation GeneratorYield with argument *itrNextObj* performs the following steps:

1. Assert: *itrNextObj* is an Object that implemented the *ItrResult* interface.
2. Let *genContext* be the running execution context.
3. Assert: *genContext* is the execution context of a generator.
4. Let *generator* be the value of the Generator component of *genContext*.
5. Set the value of *generator's* [[GeneratorState]] internal data property to `"suspendedYield"`.
6. Remove *genContext* from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.
7. Set the code evaluation state of *genContext* such that when evaluation is resumed with a Completion *resumptionValue* the following steps will be performed:
   a. Return *resumptionValue*.
   b. NOTE: This returns to the evaluation of the *YieldExpression* production that originally called this abstract operation.
8. Resume the suspended evaluation of *genContext* using NormalCompletion(*value*) as the result of the operation that suspended it. Let *result* be the value returned by the resumed computation.
9. Assert: When we return here, *gebContext* has already been removed from the execution context stack and *methodContext* is the currently running execution context.
10. Return NormalCompletion(*itrNextObj*).
11. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of *genContext*.

Deleted: d

15.19.4.3.4 CreateItrResultObject (value, done)

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

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

15.19.4.3.5 GetIterator (obj)

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

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

15.19.4.3.6 IteratorNext (iterator, value)

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

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

15.19.4.3.7 IteratorComplete (itrResult)

The abstract operation IteratorComplete with argument *itrResult* performs the following steps:

6. Assert: Type(*itrResult*) is Object.
7. Let *done* be the result of Get(*itrResult*, `"done"`).
8. Return ToBoolean(*done*).

15.19.4.3.8 IteratorValue (itrResult)

The abstract operation IteratorValue with argument *itrResult* performing the following steps:

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

15.19.4.3.9 CreateEmptyIterator ( )

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

1. Let *obj* be the result of performing ObjectCreate().
2. Let *emptyNextMethod* be the result of CreateBuiltinFunction with steps defined below.
3. Perform CreateOwnDataProperty(*obj*, `"next"`, *emptyNextMethod*).
4. Return *obj*.

An *emptyNextMethod* method performs the following steps:

1. Let *result* be the result of performing CreateItrResultObjectCreate(**undefined**, **true**).

Deleted: ing

2. Return *result*.

16 Errors

An implementation must report most errors at the time the relevant ECMAScript language construct is evaluated. An *early error* is an error that can be detected and reported prior to the evaluation of any construct in the *Script* containing the error. An implementation must report early errors in a *Script* prior to the first evaluation of that *Script*. Early errors in **eval** code are reported at the time **eval** is called but prior to evaluation of any construct within the **eval** code. All errors that are not early errors are runtime errors.

An implementation must treat as an early error any instance of an early error that is specified in a static

An implementation shall not treat other kinds of errors as early errors even if the compiler can prove that a construct cannot execute without error under any circumstances. An implementation may issue an early warning in such a case, but it should not report the error until the relevant construct is actually executed.

An implementation shall report all errors as specified, except for the following:

- An implementation may extend script syntax and regular expression pattern or flag syntax. To permit this, all operations (such as calling **eval**, using a regular expression literal, or using the **Function** or **RegExp** constructor) that are allowed to throw **SyntaxError** are permitted to exhibit implementation-defined behaviour instead of throwing **SyntaxError** when they encounter an implementation-defined extension to the script syntax or regular expression pattern or flag syntax.
- An implementation may provide additional types, values, objects, properties, and functions beyond those described in this specification. This may cause constructs (such as looking up a variable in the global scope) to have implementation-defined behaviour instead of throwing an error (such as **ReferenceError**).
- An implementation may define behaviour other than throwing **RangeError** for **toFixed**, **toExponential**, and **toPrecision** when the *fractionDigits* or *precision* argument is outside the specified range.

Deleted: *Program*

Deleted: *Program*

Deleted: *Program*

Deleted: of the following kinds of errors as an early error:

Deleted: ¶
Any syntax error.¶
Attempts to define an *ObjectLiteral* that has multiple **ge**
property assignments with the same name or multiple
property assignments with the same name.¶
Attempts to define an *ObjectLiteral* that has both a data
property assignment and a **get** or **set** property
assignment with the same name.¶
Errors in regular expression literals that are not
implementation-defined syntax extensions.¶
Attempts in strict mode code to define an *ObjectLiteral*
that has multiple data property assignments with the
same name.¶
The occurrence of a *WithStatement* in strict mode code.
The occurrence of an *Identifier* value appearing more t
once within a *FormalParameterList* of an individual strict
mode *FunctionDeclaration* or *FunctionExpression*.¶
Improper uses of **return**, **break**, and **continue**.

Comment [AWB15234]: TODO: need to verify that
actually have replicitly overed all the PutValue cases
there the deleted item would have applied to

Deleted: <#>Attempts to call PutValue on any valu
for which an early determination can be made that th
value is not a Reference (for example, executing the
assignment statement 3=4).¶

Deleted: program

Deleted: program

Annex A
(informative)

Grammar Summary

A.1 Lexical Grammar

SourceCharacter **::** *See clause 6*
any Unicode code unit

InputElementDiv **::** *See clause 7*
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp **::** *See clause 7*
WhiteSpace
LineTerminator
Comment
Token
RegularExpressionLiteral

WhiteSpace **::** *See 7.2*
<TAB>
<VT>
<FF>
<SP>
<NBSP>
<BOM>
<USP>

LineTerminator **::** *See 7.3*
<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence **::** *See 7.3*
<LF>
<CR> [lookahead ∉ <LF>]
<LS>
<PS>
<CR> <LF>

Comment **::** *See 7.4*
MultiLineComment
SingleLineComment

MultiLineComment **::** See 7.4
  `/*` *MultiLineCommentChars*$_{\text{opt}}$ `*/`

MultiLineCommentChars **::** See 7.4
  MultiLineNotAsteriskChar MultiLineCommentChars$_{\text{opt}}$
  `*` *PostAsteriskCommentChars*$_{\text{opt}}$

PostAsteriskCommentChars **::** See 7.4
  MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{\text{opt}}$
  `*` *PostAsteriskCommentChars*$_{\text{opt}}$

MultiLineNotAsteriskChar **::** See 7.4
  SourceCharacter **but not** `*`

MultiLineNotForwardSlashOrAsteriskChar **::** See 7.4
  SourceCharacter **but not one of** `/` **or** `*`

SingleLineComment **::** See 7.4
  `//` *SingleLineCommentChars*$_{\text{opt}}$

SingleLineCommentChars **::** See 7.4
  SingleLineCommentChar SingleLineCommentChars$_{\text{opt}}$

SingleLineCommentChar **::** See 7.4
  SourceCharacter **but not** *LineTerminator*

Token **::** See 7.5
  IdentifierName
  Punctuator
  NumericLiteral
  StringLiteral

Identifier **::** See 7.6
  IdentifierName **but not** *ReservedWord*

IdentifierName **::** See 7.6
  IdentifierStart
  IdentifierName IdentifierPart

IdentifierStart **::** See 7.6
  UnicodeLetter
  `$`
  `_`
  `\` *UnicodeEscapeSequence*

IdentifierPart **::** *See 7.6*
IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
<ZWNJ>
<ZWJ>

UnicodeLetter **::** *See 7.6*
any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Nl)”.

UnicodeCombiningMark **::** *See 7.6*
any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

UnicodeDigit **::** *See 7.6*
any character in the Unicode category “Decimal number (Nd)”

UnicodeConnectorPunctuation **::** *See 7.6*
any character in the Unicode category “Connector punctuation (Pc)”

ReservedWord **::** *See 7.6.1*
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

Keyword **:: one of** *See 7.6.1.1*

`break`	**`do`**	**`instanceof`**	**`typeof`**
`case`	**`else`**	**`new`**	**`var`**
`catch`	**`finally`**	**`return`**	**`void`**
`continue`	**`for`**	**`switch`**	**`while`**
`debugger`	**`function`**	**`this`**	**`with`**
`default`	**`if`**	**`throw`**	
`delete`	**`in`**	**`try`**	

FutureReservedWord **:: one of** *See 7.6.1.2*

`class`	**`enum`**	**`extends`**	**`super`**
`const`	**`export`**	**`import`**	

The following tokens are also considered to be *FutureReservedWords* when parsing strict mode code (see 10.1.1).

`implements`	**`let`**	**`private`**	**`public`**
`interface`	**`package`**	**`protected`**	**`static`**
`yield`			

Punctuator **:: one of** *See 7.7*

```
{      }      (      )      [      ]
.      ;      ,      <      >      <=
>=     ==     !=     ===    !==
+      -      *      %      ++     --
<<     >>     >>>    &      |      ^
!      ~      &&     ||     ?      :
=      +=     -=     *=     %=     <<=
>>=    >>>=   &=     |=     ^=
```

DivPunctuator **:: one of** *See 7.7*

`/` **`/=`**

Literal **::** *See 7.8*

NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral

NullLiteral **::** *See 7.8.1*

`null`

BooleanLiteral **::** *See 7.8.2*

`true`
`false`

NumericLiteral **::** *See 7.8.3*

DecimalLiteral
HexIntegerLiteral

DecimalLiteral **::** *See 7.8.3*

DecimalIntegerLiteral **`.`** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
`.` *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart$_{opt}$

DecimalIntegerLiteral **::** *See 7.8.3*

`0`
NonZeroDigit DecimalDigits$_{opt}$

DecimalDigits **::** *See 7.8.3*

DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **:: one of** *See 7.8.3*

`0 1 2 3 4 5 6 7 8 9`

NonZeroDigit **:: one of** *See 7.8.3*
`1 2 3 4 5 6 7 8 9`

ExponentPart **::** *See 7.8.3*
ExponentIndicator SignedInteger

ExponentIndicator **:: one of** *See 7.8.3*
`e E`

SignedInteger **::** *See 7.8.3*
DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

HexIntegerLiteral **::** *See 7.8.3*
`0x` *HexDigit*
`0X` *HexDigit*
HexIntegerLiteral HexDigit

HexDigit **:: one of** *See 7.8.3*
`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

StringLiteral **::** *See 7.8.4*
`"` $DoubleStringCharacters_{opt}$ **`"`**
`'` $SingleStringCharacters_{opt}$ **`'`**

DoubleStringCharacters **::** *See 7.8.4*
DoubleStringCharacter $DoubleStringCharacters_{opt}$

SingleStringCharacters **::** *See 7.8.4*
SingleStringCharacter $SingleStringCharacters_{opt}$

DoubleStringCharacter **::** *See 7.8.4*
SourceCharacter **but not one of `"` or `\` or** *LineTerminator*
`\` *EscapeSequence*
LineContinuation

SingleStringCharacter **::** *See 7.8.4*
SourceCharacter **but not one of `'` or `\` or** *LineTerminator*
`\` *EscapeSequence*
LineContinuation

LineContinuation **::** *See 7.8.4*
`\` *LineTerminatorSequence*

EscapeSequence **::** *See 7.8.4*
CharacterEscapeSequence
`0` [lookahead ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **::** *See 7.8.4*
SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter **:: one of** *See 7.8.4*
`' " \ b f n r t v`

NonEscapeCharacter **::** See 7.8.4
SourceCharacter **but not one of** *EscapeCharacter* **or** *LineTerminator*

EscapeCharacter **::** See 7.8.4
SingleEscapeCharacter
DecimalDigit
`x`
`u`

HexEscapeSequence **::** See 7.8.4
`x` *HexDigit HexDigit*

UnicodeEscapeSequence **::** See 7.8.4
`u` *HexDigit HexDigit HexDigit HexDigit*

RegularExpressionLiteral **::** See 7.8.5
/ *RegularExpressionBody* **/** *RegularExpressionFlags*

RegularExpressionBody **::** See 7.8.5
RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars **::** See 7.8.5
[empty]
RegularExpressionChars RegularExpressionChar

RegularExpressionFirstChar **::** See 7.8.5
RegularExpressionNonTerminator **but not one of `*` or `\` or `/` or `[`**
RegularExpressionBackslashSequence
RegularExpressionClass

RegularExpressionChar **::** See 7.8.5
RegularExpressionNonTerminator **but not `\` or `/` or `[`**
RegularExpressionBackslashSequence
RegularExpressionClass

RegularExpressionBackslashSequence **::** See 7.8.5
`\` *RegularExpressionNonTerminator*

RegularExpressionNonTerminator **::** See 7.8.5
SourceCharacter **but not** *LineTerminator*

RegularExpressionClass **::** See 7.8.5
`[` *RegularExpressionClassChars* **`]`**

RegularExpressionClassChars **::** See 7.8.5
[empty]
RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar **::** See 7.8.5
RegularExpressionNonTerminator **but not `]` or `\`**
RegularExpressionBackslashSequence

RegularExpressionFlags **::** *See 7.8.5*
[empty]
RegularExpressionFlags IdentifierPart

A.2 Number Conversions

StringNumericLiteral **:::** *See 9.1.3.1*
StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ StrNumericLiteral StrWhiteSpace$_{opt}$

StrWhiteSpace **:::** *See 9.1.3.1*
StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **:::** *See 9.1.3.1*
WhiteSpace
LineTerminator

StrNumericLiteral **:::** *See 9.1.3.1*
StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral **:::** *See 9.1.3.1*
StrUnsignedDecimalLiteral
+ *StrUnsignedDecimalLiteral*
- *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral **:::** *See 9.1.3.1*
`Infinity`
DecimalDigits **.** *DecimalDigits$_{opt}$ ExponentPart$_{opt}$*
. *DecimalDigits ExponentPart$_{opt}$*
DecimalDigits ExponentPart$_{opt}$

DecimalDigits **:::** *See 9.1.3.1*
DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **::: one of** *See 9.1.3.1*
`0 1 2 3 4 5 6 7 8 9`

ExponentPart **:::** *See 9.1.3.1*
ExponentIndicator SignedInteger

ExponentIndicator **::: one of** *See 9.1.3.1*
`e E`

SignedInteger **:::** *See 9.1.3.1*
DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

HexIntegerLiteral ::: See 9.1.3.1
    0x HexDigit
    0X HexDigit
    HexIntegerLiteral HexDigit

HexDigit ::: one of See 9.1.3.1
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

A.3 Expressions

PrimaryExpression : See 11.1
    this
    Identifier
    Literal
    ArrayLiteral
    ObjectLiteral
    (Expression)

ArrayLiteral : See 11.1.4
    [$Elision_{opt}$]
    [ElementList]
    [ElementList , $Elision_{opt}$]

ElementList : See 11.1.4
    $Elision_{opt}$ AssignmentExpression
    ElementList , $Elision_{opt}$ AssignmentExpression

Elision : See 11.1.4
    ,
    Elision ,

ObjectLiteral : See 11.1.5
    { }
    { PropertyDefinitionList }
    { PropertyDefinitionList , }

Deleted: PropertyNameAndValueList
Deleted: PropertyNameAndValueList

PropertyDefinitionList : See 11.1.5
    PropertyDefinition
    PropertyDefinitionList , PropertyDefinition

Deleted: PropertyNameAndValueList
Deleted: PropertyAssignment
Deleted: PropertyNameAndValueList
Deleted: PropertyAssignment
Deleted: PropertyAssignment

PropertyDefinition : See 11.1.5
    PropertyName : AssignmentExpression
    get PropertyName ( ) { FunctionBody }
    set PropertyName (PropertySetParameterList) { FunctionBody }

PropertyName : See 11.1.5
    IdentifierName
    StringLiteral
    NumericLiteral

PropertySetParameterList : See 11.1.5
    Identifier

MemberExpression **:** See 11.2
PrimaryExpression
FunctionExpression
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
new *MemberExpression Arguments*

NewExpression **:** See 11.2
MemberExpression
new *NewExpression*

CallExpression **:** See 11.2
MemberExpression Arguments
CallExpression Arguments
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*

Arguments **:** See 11.2
( )
(*ArgumentList* **)**

ArgumentList **:** See 11.2
AssignmentExpression
ArgumentList **,** *AssignmentExpression*

LeftHandSideExpression **:** See 11.2
NewExpression
CallExpression

PostfixExpression **:** See 11.3
LeftHandSideExpression
LeftHandSideExpression [no *LineTerminator* here] **++**
LeftHandSideExpression [no *LineTerminator* here] **--**

UnaryExpression **:** See 11.4
PostfixExpression
delete *UnaryExpression*
void *UnaryExpression*
typeof *UnaryExpression*
++ *UnaryExpression*
-- *UnaryExpression*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*

MultiplicativeExpression **:** See 11.5
UnaryExpression
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

AdditiveExpression **:** See 11.6
MultiplicativeExpression
AdditiveExpression **+** *MultiplicativeExpression*
AdditiveExpression **-** *MultiplicativeExpression*

ShiftExpression **:** See 11.7
AdditiveExpression
ShiftExpression **<<** *AdditiveExpression*
ShiftExpression **>>** *AdditiveExpression*
ShiftExpression **>>>** *AdditiveExpression*

RelationalExpression **:** See 11.8
ShiftExpression
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*
RelationalExpression **`instanceof`** *ShiftExpression*
RelationalExpression **`in`** *ShiftExpression*

RelationalExpressionNoIn **:** See 11.8
ShiftExpression
RelationalExpressionNoIn **<** *ShiftExpression*
RelationalExpressionNoIn **>** *ShiftExpression*
RelationalExpressionNoIn **<=** *ShiftExpression*
RelationalExpressionNoIn **>=** *ShiftExpression*
RelationalExpressionNoIn **`instanceof`** *ShiftExpression*

EqualityExpression **:** See 11.9
RelationalExpression
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*

EqualityExpressionNoIn **:** See 11.9
RelationalExpressionNoIn
EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

BitwiseANDExpression **:** See 11.10
EqualityExpression
BitwiseANDExpression **&** *EqualityExpression*

BitwiseANDExpressionNoIn **:** See 11.10
EqualityExpressionNoIn
BitwiseANDExpressionNoIn **&** *EqualityExpressionNoIn*

BitwiseXORExpression **:** See 11.10
    BitwiseANDExpression
    BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseXORExpressionNoIn **:** See 11.10
    BitwiseANDExpressionNoIn
    BitwiseXORExpressionNoIn **^** *BitwiseANDExpressionNoIn*

BitwiseORExpression **:** See 11.10
    BitwiseXORExpression
    BitwiseORExpression **|** *BitwiseXORExpression*

BitwiseORExpressionNoIn **:** See 11.10
    BitwiseXORExpressionNoIn
    BitwiseORExpressionNoIn **|** *BitwiseXORExpressionNoIn*

LogicalANDExpression **:** See 11.11
    BitwiseORExpression
    LogicalANDExpression **&&** *BitwiseORExpression*

LogicalANDExpressionNoIn **:** See 11.11
    BitwiseORExpressionNoIn
    LogicalANDExpressionNoIn **&&** *BitwiseORExpressionNoIn*

LogicalORExpression **:** See 11.11
    LogicalANDExpression
    LogicalORExpression **||** *LogicalANDExpression*

LogicalORExpressionNoIn **:** See 11.11
    LogicalANDExpressionNoIn
    LogicalORExpressionNoIn **||** *LogicalANDExpressionNoIn*

ConditionalExpression **:** See 11.12
    LogicalORExpression
    LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

ConditionalExpressionNoIn **:** See 11.12
    LogicalORExpressionNoIn
    LogicalORExpressionNoIn **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

AssignmentExpression **:** See 11.13
    ConditionalExpression
    LeftHandSideExpression **=** *AssignmentExpression*
    LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn **:** See 11.13
    ConditionalExpressionNoIn
    LeftHandSideExpression **=** *AssignmentExpressionNoIn*
    LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

AssignmentOperator **: one of** See 11.13

`*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `>>>=` `&=` `^=` `|=`

Expression **:** See 11.14

AssignmentExpression
Expression **,** *AssignmentExpression*

ExpressionNoIn **:** See 11.14

AssignmentExpressionNoIn
ExpressionNoIn **,** *AssignmentExpressionNoIn*

A.4 Statements

Statement **:** See clause 12

Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement

Block **:** See 12.1

{ *StatementList*$_{opt}$ **}**

StatementList **:** See 12.1

Statement
StatementList Statement

VariableStatement **:** See 12.2

`var` *VariableDeclarationList* **;**

VariableDeclarationList **:** See 12.2

VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclarationListNoIn **:** See 12.2

VariableDeclarationNoIn
VariableDeclarationListNoIn **,** *VariableDeclarationNoIn*

VariableDeclaration **:** See 12.2

Identifier Initialiser$_{opt}$

VariableDeclarationNoIn **:** See 12.2

Identifier InitialiserNoIn$_{opt}$

Initialiser **:** *See 12.2*
  = *AssignmentExpression*

InitialiserNoIn **:** *See 12.2*
  = *AssignmentExpressionNoIn*

EmptyStatement **:** *See 12.3*
  ;

ExpressionStatement **:** *See 12.4*
  [lookahead ∉ {**{**, **function**}] *Expression* **;**

IfStatement **:** *See 12.5*
  if (*Expression* **)** *Statement* **else** *Statement*
  if (*Expression* **)** *Statement*

IterationStatement **:** *See 12.6*
  do *Statement* **while (** *Expression* **);**
  while (*Expression* **)** *Statement*
  for (*ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
  for (var *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
  for (*LeftHandSideExpression* **in** *Expression* **)** *Statement*
  for (var *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*

ContinueStatement **:** *See 12.7*
  continue ;
  continue [no *LineTerminator* here] *Identifier* **;**

BreakStatement **:** *See 12.8*
  break ;
  break [no *LineTerminator* here] *Identifier* **;**

ReturnStatement **:** *See 12.9*
  return ;
  return [no *LineTerminator* here] *Expression* **;**

WithStatement **:** *See 12.10*
  with (*Expression* **)** *Statement*

SwitchStatement **:** *See 12.11*
  switch (*Expression* **)** *CaseBlock*

CaseBlock **:** *See 12.11*
  { *CaseClauses*$_{opt}$ **}**
  { *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

CaseClauses **:** *See 12.11*
  CaseClause
  CaseClauses CaseClause

CaseClause **:** See 12.11
  `case` *Expression* **`:`** *StatementList*$_{opt}$

DefaultClause **:** See 12.11
  `default` **`:`** *StatementList*$_{opt}$

LabelledStatement **:** See 12.12
  Identifier **`:`** *Statement*

ThrowStatement **:** See 12.13
  `throw` [no *LineTerminator* here] *Expression* **`;`**

TryStatement **:** See 12.14
  `try` *Block Catch*
  `try` *Block Finally*
  `try` *Block Catch Finally*

Catch **:** See 12.14
  `catch (` *Identifier* **`)`** *Block*

Finally **:** See 12.14
  `finally` *Block*

DebuggerStatement **:** See 12.15
  `debugger ;`

A.5 Functions and Scripts

Deleted: Programs

FunctionDeclaration **:** See clause 13
  `function` *Identifier* **`(`** *FormalParameterList*$_{opt}$ **`)`** **`{`** *FunctionBody* **`}`**

FunctionExpression **:** See clause 13
  `function` *Identifier*$_{opt}$ **`(`** *FormalParameterList*$_{opt}$ **`)`** **`{`** *FunctionBody* **`}`**

FormalParameterList **:** See clause 13
  Identifier
  FormalParameterList **`,`** *Identifier*

FunctionBody **:** See clause 13
  SourceElements$_{opt}$

Program **:** See clause 14
  SourceElements$_{opt}$

SourceElements **:** See clause 14
  SourceElement
  SourceElements SourceElement

SourceElement **:** *See clause 14*
Statement
FunctionDeclaration

A.6 Universal Resource Identifier Character Classes

uri **:::** *See 15.1.3*
uriCharacters$_{opt}$

uriCharacters **:::** *See 15.1.3*
uriCharacter uriCharacters$_{opt}$

uriCharacter **:::** *See 15.1.3*
uriReserved
uriUnescaped
uriEscaped

uriReserved **::: one of** *See 15.1.3*
`;  /  ?  :  @  &  =  +  $  ,`

uriUnescaped **:::** *See 15.1.3*
uriAlpha
DecimalDigit
uriMark

uriEscaped **:::** *See 15.1.3*
`%` *HexDigit HexDigit*

uriAlpha **::: one of** *See 15.1.3*
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

uriMark **::: one of** *See 15.1.3*
`-  _  .  !  ~  *  '  (  )`

A.7 Regular Expressions

Pattern **::** *See 15.10.1*
Disjunction

Disjunction **::** *See 15.10.1*
Alternative
Alternative **|** *Disjunction*

Alternative **::** *See 15.10.1*
[empty]
Alternative Term

Term **::** *See 15.10.1*
Assertion
Atom
Atom Quantifier

Assertion **::** *See 15.10.1*

^
$
\ b
\ B
(? = *Disjunction* **)**
(? ! *Disjunction* **)**

Quantifier **::** *See 15.10.1*

QuantifierPrefix
QuantifierPrefix **?**

QuantifierPrefix **::** *See 15.10.1*

*****
+
?
{ *DecimalDigits* **}**
{ *DecimalDigits* **, }**
{ *DecimalDigits* **,** *DecimalDigits* **}**

Atom **::** *See 15.10.1*

PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(? : *Disjunction* **)**

PatternCharacter :: *See 15.10.1*

SourceCharacter **but not one of-**
^ $ \ . * + ? ( ) [ ] { } |

AtomEscape **::** *See 15.10.1*

DecimalEscape
CharacterEscape
CharacterClassEscape

CharacterEscape **::** *See 15.10.1*

ControlEscape
c *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

ControlEscape **:: one of** *See 15.10.1*

f n r t v

ControlLetter **:: one of** *See 15.10.1*

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

IdentityEscape **::** See 15.10.1
SourceCharacter **but not** *IdentifierPart*
<ZWJ>
<ZWNJ>

DecimalEscape **::** See 15.10.1
DecimalIntegerLiteral [lookahead ∉ *DecimalDigit*]

CharacterClassEscape **::** **one of** See 15.10.1
`d   D   s   S   w   W`

CharacterClass **::** See 15.10.1
`[` [lookahead ∉ {**`^`**}] *ClassRanges* **`]`**
`[` **`^`** *ClassRanges* **`]`**

ClassRanges **::** See 15.10.1
[empty]
NonemptyClassRanges

NonemptyClassRanges **::** See 15.10.1
ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom **`-`** *ClassAtom ClassRanges*

NonemptyClassRangesNoDash **::** See 15.10.1
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash **`-`** *ClassAtom ClassRanges*

ClassAtom **::** See 15.10.1
`-`
ClassAtomNoDash

ClassAtomNoDash **::** See 15.10.1
SourceCharacter **but not one of `\` or `]` or `-`**
`\` *ClassEscape*

ClassEscape **::** See 15.10.1
DecimalEscape
`b`
CharacterEscape
CharacterClassEscape

A.8 JSON

A.8.1 JSON Lexical Grammar

JSONWhiteSpace **::** See 15.12.1.1
<TAB>
<CR>

<LF>
<SP>

JSONString **::** *See 15.12.1.1*
" *JSONStringCharacters*$_{opt}$ **"**

JSONStringCharacters **::** *See 15.12.1.1*
JSONStringCharacter JSONStringCharacters$_{opt}$

JSONStringCharacter **::** *See 15.12.1.1*
SourceCharacter **but not one of " or \ or** *U+0000* **through** *U+001F*
**** *JSONEscapeSequence*

JSONEscapeSequence **::** *See 15.12.1.1*
JSONEscapeCharacter
UnicodeEscapeSequence

JSONEscapeCharacter **:: one of** *See 15.12.1.1*
" / \ b f n r t

JSONNumber **::** *See 15.12.1.1*
-$_{opt}$ *DecimalIntegerLiteral JSONFraction*$_{opt}$ *ExponentPart*$_{opt}$

JSONFraction **::** *See 15.12.1.1*
. *DecimalDigits*

JSONNullLiteral **::** *See 15.12.1.1*
NullLiteral

JSONBooleanLiteral **::** *See 15.12.1.1*
BooleanLiteral

A.8.2 JSON Syntactic Grammar

JSONText **:** *See 15.12.1.2*
JSONValue

JSONValue **:** *See 15.12.1.2*
JSONNullLiteral
JSONBooleanLiteral
JSONObject
JSONArray
JSONString
JSONNumber

JSONObject **:** *See 15.12.1.2*
{ }
{ *JSONMemberList* **}**

JSONMember **:** *See 15.12.1.2*
JSONString **:** *JSONValue*

JSONMemberList **:** *See 15.12.1.2*
JSONMember
JSONMemberList **,** *JSONMember*

JSONArray **:** *See 15.12.1.2*
[ ]
[*JSONElementList* **]**

JSONElementList **:** *See 15.12.1.2*
JSONValue
JSONElementList **,** *JSONValue*

Annex B
(normative)

Additional ECMAScript Features for Web Browsers

The ECMAScript language syntax and semantics defined in this annex are required when the ECMAScript host is a web browser. The content of this annex is normative but optional if the ECMAScript host is not a web browser.

B.1 Additional Syntax

B.1.1 Numeric Literals

The syntax and semantics of 7.8.3 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

Static Semantics

- The MV of *LegacyOctalIntegerLiteral* **::** `0` *OctalDigit* is the MV of *OctalDigit*.
- The MV of *LegacyOctalIntegerLiteral* **::** *LegacyOctalIntegerLiteral OctalDigit* is (the MV of *LegacyOctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.

B.1.2 String Literals

The syntax and semantics of 7.8.4 is extended as follows except that this extension is not allowed for strict mode code:

Syntax

EscapeSequence **::**
  CharacterEscapeSequence
  OctalEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence

OctalEscapeSequence **::**
  OctalDigit [lookahead ∉ *DecimalDigit*]
  ZeroToThree OctalDigit [lookahead ∉ *DecimalDigit*]
  FourToSeven OctalDigit
  ZeroToThree OctalDigit OctalDigit

ZeroToThree **:: one of**
  `0   1   2   3`

Comment [AW235]: Need to add new material that i not in previous verions of annex B

Deleted: informative

Deleted: Compatibility

Deleted: is

Deleted: s

Deleted: It

Deleted: <#>Past editions of ECMAScript have included additional syntax and semantics for specifying octal literals and octal escape sequence These have been removed from this edition of ECMAScript. This non-normative annex presents uniform syntax and semantics for octal literals and octal escape sequences for compatibility with som older ECMAScript programs.¶

Deleted: can be

Deleted: *OctalDigit* **:: one of¶**
`0 1 2 3 4 5 6 7¶`

Deleted: <#>The MV of *NumericLiteral* **::** *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.¶
<#>The MV of *OctalDigit* **::** `0` is 0.¶
<#>The MV of *OctalDigit* **::** `1` is 1.¶
<#>The MV of *OctalDigit* **::** `2` is 2.¶
<#>The MV of *OctalDigit* **::** `3` is 3.¶
<#>The MV of *OctalDigit* **::** `4` is 4.¶
<#>The MV of *OctalDigit* **::** `5` is 5.¶
<#>The MV of *OctalDigit* **::** `6` is 6.¶
<#>The MV of *OctalDigit* **::** `7` is 7.¶

Deleted: can be

FourToSeven :: **one of**
`4 5 6 7`

Static Semantics

- The CV of *EscapeSequence* :: *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *OctalEscapeSequence* :: *OctalDigit* [lookahead ∉ *DecimalDigit*] is the character whose code unit value is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *ZeroToThree OctalDigit* [lookahead ∉ *DecimalDigit*] is the character whose code unit value is (8 times the MV of the *ZeroToThree*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *FourToSeven OctalDigit* is the character whose code unit value is (8 times the MV of the *FourToSeven*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *ZeroToThree OctalDigit OctalDigit* is the character whose code unit value is (64 (that is, 8^2) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree* :: `0` is 0.
- The MV of *ZeroToThree* :: `1` is 1.
- The MV of *ZeroToThree* :: `2` is 2.
- The MV of *ZeroToThree* :: `3` is 3.
- The MV of *FourToSeven* :: `4` is 4.
- The MV of *FourToSeven* :: `5` is 5.
- The MV of *FourToSeven* :: `6` is 6.
- The MV of *FourToSeven* :: `7` is 7.

B.1.3 HTML-like Comments

TODO See http://javascript.spec.whatwg.org/#comment-syntax

B.2 Additional Properties

When the ECMAScript host is a web browser the following additional properties of the standard built-in objects are defined.

B.2.1 Additional Properties of the Global Object

B.2.1.1 escape (string)

The **escape** function is a property of the global object. It computes a new version of a String value in which certain characters have been replaced by a hexadecimal escape sequence.

For those characters being replaced whose code unit value is **0xFF** or less, a two-digit escape sequence of the form **%***xx* is used. For those characters being replaced whose code unit value is greater than **0xFF**, a four-digit escape sequence of the form **%u***xxxx* is used.

When the **escape** function is called with one argument *string*, the following steps are taken:

1. Let *string* be ToString(*string*).
2. ReturnIfAbrupt(*string*).
3. Let *length* be the number of code units in *string*.
4. Let *R* be the empty string.
5. Let *k* be 0.
6. Repeat, while *k* < *length*,
   a. Let *char* be the code unit (represented as a 16-bit unsigned integer) at position *k* within *string*.
   b. If *char* is the code point of one of the 69 nonblank characters **"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./"** then,
      i. Let *S* be a String containing the single character *char*.

Deleted: Some implementations of
Deleted: have included
Deleted: these
Deleted: for some
Deleted: native
Deleted: This non-normative annex suggest uniform semantics for such properties withou making the properties or their semantics part this standard.
Deleted: Call
Deleted: Compute
Deleted: characters
Deleted: Result(1)
Deleted: If
Deleted: equals
Deleted: Result(2)
Deleted: return *R*.
Deleted: Get
Deleted: character
Deleted: Result(1)
Deleted: *Result(6)*
Deleted: "
Deleted: "
Deleted: go to step 13.

c. Else if *char* ≥ 256,
   i. Let *S* be a String containing six characters "%u*wxyz*" where *wxyz* are four hexadecimal digits encoding the value of *char*.
d. Else, *char* < 256
   i. Let *S* be a String containing three characters "%*xy*" where *xy* are two hexadecimal digits encoding the value of *char*.
e. Let *R* be a new String value computed by concatenating the previous value of *R* and *S*.
f. Increase *k* by 1.

7. Return *R*.

NOTE The encoding is partly based on the encoding described in RFC 1738, but the entire encoding specified in this standard is described above without regard to the contents of RFC 1738. This encoding does not reflect changes to RFC 1738 made by RFC 3986.

B.2.1.2 unescape (string)

The **unescape** function is a property of the global object. It computes a new version of a String value in which each escape sequence of the sort that might be introduced by the **escape** function is replaced with the character that it represents.

When the **unescape** function is called with one argument *string*, the following steps are taken:

1. Let *string* be ToString(*string*).
2. ReturnIfAbrupt(*string*).
3. Let *length* be the number of code units in *string*.
4. Let *R* be the empty String.
5. Let *k* be 0.
6. Repeat, while *k* ≠ *length*
   a. Let *c* be the code unit at position *k* within *string*.
   b. If *c* is %,
      i. If *k* ≤ *length*–6 and the code unit at position *k*+1 within *string* is u and the four code units at positions *k*+2, *k*+3, *k*+4, and *k*+5 within *string* are all hexadecimal digits, then
         1. Let *c* be the code unit whose value is the integer represented by the four hexadecimal digits at positions *k*+2, *k*+3, *k*+4, and *k*+5 within *string*.
         2. Increase *k* by 5.
      ii. Else if *k* ≤ *length*–3 and the two code units at positions *k*+1 and *k*+2 within *string* are both hexadecimal digits, then.
         1. Let *c* be the code unit whose value is the integer represented by two zeroes plus the two hexadecimal digits at positions *k*+1 and *k*+2 within *string*.
         2. Increase *k* by 2.
   c. Let *R* be a new String value computed by concatenating the previous value of *R* and *c*.
   d. Increase *k* by 1.
7. Return *R*.

B.2.2 Additional Properties of the Object.prototype Object

B.2.2.1 Object.prototype.__proto__

Object.prototype.__proto__ is an accessor property with attributes { [[Enumerable]]: **false**, [[Configurable]]: **true** }. The [[Get]] and [[Set]] attributes are defined as follows

16.1.1.1.1 get Object.prototype.__proto__

The value of the [[Get]] attribute is a built-in function that requires no arguments. It performs the following steps:

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, then throw a **TypeError** exception.
3. Return the result of calling the [[GetInheritance]] internal method of *O*.

Deleted: If …lse if *char*Result(6)
Deleted: ,
Deleted: is less than… 256, go to step 11.
Deleted: "…u*wxyz*"…where *wxyz* are four hexadeci
Deleted: Go to step 14.
Deleted: "…*xy*"…where *xy* are two hexadecimal digi
Deleted: <#>Go to step 14.¶
<#>Let *S* be a String containing the single character Result(6).¶
Deleted: Go to step 5
Deleted: Call
Deleted: Compute …he number of code units characters
Deleted: If …epeat, while *k* ≠ equals …*length*Result(2),
Deleted: character …t position *k* within *string*Result(1)
Deleted: not ..., go to step 18.
Deleted: is greater than…*length*Result(2)…6, go to ste
Deleted: character…whose code unit …alue is the
Deleted: <#>Go to step 18.¶
Deleted: character…whose code unit …alue is the
Deleted: Go to step 5.

16.1.1.1.2 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 the **this** value.
2. If Type(*O*) is not Object, then throw a **TypeError** exception.
3. If Type(*proto*) is neither Object or Null, then throw a **TypeError** exception.
4. Let *status* be the result of calling the [[SetInheritance]] internal method of *O* with argument *proto*.
5. ReturnIfAbrupt(*status*).
6. If *status* is **false**, then throw a **TypeError** exception.
7. Return *proto*.

B.2.3 Additional Properties of the String.prototype Object

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

The `substr` method takes two arguments, *start* and *length*, and returns a substring of the result of converting the this object to a String, starting from character position *start* and running for *length* characters (or through the end of the String if *length* is **undefined**). If *start* is negative, it is treated as (*sourceLength*+*start*) where *sourceLength* is the length of the String. The result is a String value, not a String object. The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. Let *intStart* be ToInteger(*start*).
4. ReturnIfAbrupt(*intStart*).
5. If *length* is **undefined**, let *end* be +∞; otherwise let *end* be ToInteger(*length*).
6. ReturnIfAbrupt(*end*).
7. Let *size* be the number of characters in *S*.
8. If *intStart* is negative, then let *intStart* be max(*size* + *intStart*, 0).
9. Let *resultLength* be min(max(*end*, 0), *size* – *intStart*).
10. If *resultLength* ≤ 0, return the empty String `""`.
11. Return a String containing *resultLength* consecutive characters from *S* beginning with the character at position *intStart*.

The `length` property of the `substr` method is **2**.

NOTE The `substr` function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

B.2.3.2 String.prototype.anchor (name)

When the **anchor** method is called with argument *name*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"a"`, `"name"` and *name*.

The abstract operation CreateHTML is called with arguments *string*, *tag*, *attribute*, and *value*. The arguments *tag* and *attribute* must be string values. The following steps are taken:

1. Let *str* be CheckObjectCoercible(*string*).
2. Let *S* be ToString(*str*).
3. ReturnIfAbrupt(*S*).
4. Let *p1* be the string value that is the concatenation of `"<"` and *tag*.
5. If *attribute* is not the empty String, then
   a. Let *V* be the result of performing ToString(*value*).
   b. ReturnIfAbrupt(*V*).

Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: Call
Deleted: the result of performing
Deleted: , giving it the **this** value as its argum
Deleted: Call
Deleted: use
Deleted: call
Deleted: Compute
Deleted: Result(1)
Deleted: Result(2)
Deleted: positive or zero
Deleted: use Result(2); else
Deleted: use
Deleted: Result(4)
Deleted: Result(2)
Deleted: Compute
Deleted: Result(3)
Deleted: Result(4)
Deleted: Result(5)
Deleted: Result(6)
Deleted: “”
Deleted: Result(6)
Deleted: Result(1)
Deleted: Result(5)
Deleted: ReturnIfAbrupt(
Deleted:)
Deleted: the result of performing
Deleted: *string*
Deleted: <#>If *attribute* is not the empty Stri then¶
<#>Let *V* be the result of performing ToString(*value*).¶
<#>ReturnIfAbrupt(*V*).¶

c. Let *escapedV* be the string value that is the same as *V* except that each occurrence of the character `"` (code unit value 0x0022) in *V* has been replaced with the six character sequence `"""`.

d. Let *p1* be the string value that is the concatenation of the following string values:

- *p1*
- a single space code unit 0x0020

Deleted: character

- *attribute*
- `"="`
- `'"'`
- *escapedV*

Deleted: *V*

- `'"'`

6. Let *p2* be the string value that is the concatenation of *p1* and `">"`.
7. Let *p3* be the string value that is the concatenation of *p2* and *S*.
8. Let *p4* be the string value that is the concatenation of *p2*, `"</"`, *tag*, and `">"`.

Deleted: *3*

9. Return *p4*.

Deleted: *3*

B.2.3.3 String.prototype.big ()

When the **big** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"big"`, `""` and `""`.

B.2.3.4 String.prototype.blink ()

When the **blink** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"blink"`, `""` and `""`.

Deleted: `c`

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.

Deleted: S

2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"b"`, `""` and `""`.

B.2.3.6 String.prototype.fixed ()

Deleted: A

When the **fixed** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.

Deleted: S

2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"tt"`, `""` and `""`.

B.2.3.7 String.prototype.fontcolor (color)

Deleted: A

When the **fontcolor** method is called with argument *color*, the following steps are taken:

1. Let *S* be the **this** value.

Deleted: S

2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"font"`, `"color"` and *color*.

B.2.3.8 String.prototype.fontsize (size)

Deleted: A

When the **fontsize** method is called with argument *size*, the following steps are taken:

1. Let *S* be the **this** value.

Deleted: S

2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"font"`, `"size"` and *size*.

B.2.3.9 String.prototype.italics ()

When the **italics** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"i"`, `""` and `""`.

B.2.3.10 String.prototype.link (url)

When the **link** method is called with argument *url*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"a"`, `"href"` and *url*.

B.2.3.11 String.prototype.small ()

When the **small** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"small"`, `""` and `""`.

B.2.3.12 String.prototype.strike ()

When the **strike** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"strike"`, `""` and `""`.

B.2.3.13 String.prototype.sub ()

When the **sub** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"sub"`, `""` and `""`.

B.2.3.14 String.prototype.sup ()

When the **sup** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"sup"`, `""` and `""`.

B.2.4 Additional Properties of the Date.prototype Object

B.2.4.1 Date.prototype.getYear ( )

NOTE The `getFullYear` method is preferred for nearly all purposes, because it avoids the "year 2000 problem."

When the **getYear** method is called with no arguments, the following steps are taken:

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.

Deleted: A
Deleted: S
Deleted: A
Deleted: S
Deleted: A
Deleted: S
Deleted: s
Deleted: A
Deleted: S
Deleted: return
Deleted: s
Deleted: A
Deleted: S
Deleted: return
Deleted: s
Deleted: A
Deleted: S
Deleted: return
Deleted: s

DRAFT

4. Return YearFromTime(LocalTime(*t*)) – 1900.

B.2.4.2 Date.prototype.setYear (year)

NOTE The **`setFullYear`** method is preferred for nearly all purposes, because it avoids the "year 2000 problem."

When the **setYear** method is called with one argument *year*, the following steps are taken:

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Let *y* be ToNumber(*year*).
3. If *y* is **NaN**, set the [[DateValue]] internal data property of this Date object to **NaN** and return **NaN**.
4. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99 then let *yyyy* be ToInteger(*y*) + 1900. Otherwise, let *yyyy* be *y*.
5. Let *d* be MakeDay(*yyyy*, MonthFromTime(*t*), DateFromTime(*t*)).
6. Let *date* be UTC(MakeDate(*d*, TimeWithinDay(*t*))).
7. Set the [[DateValue]] internal data property of this Date object to TimeClip(*date*).
8. Return the value of the [[DateValue]] internal data property of this Date object.

Deleted: Call
Deleted: Result(2)
Deleted: PrimitiveValue
Deleted: the **this** value
Deleted: Result(2)
Deleted: Result(2)
Deleted: Result(4)
Deleted: is
Deleted: Result(2)
Deleted: Result(4)
Deleted: is
Deleted: Result(2)
Deleted: Compute
Deleted: Result(4)
Deleted: Compute
Deleted: Result(5)
Deleted: PrimitiveValue
Deleted: the **this** value
Deleted: Result(6)
Deleted: PrimitiveValue
Deleted: the **this** value

B.2.4.3 Date.prototype.toGMTString ( )

NOTE The property **`toUTCString`** is preferred. The **`toGMTString`** property is provided principally for compatibility with old code. It is recommended that the **`toUTCString`** property be used in new ECMAScript code.

The Function object that is the initial value of **`Date.prototype.toGMTString`** is the same Function object that is the initial value of **`Date.prototype.toUTCString`**.

B.2.5 Additional Properties of the RegExp.prototype Object

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

When the **compile** method is called with no arguments, 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 data property, then
   a. Throw a **TypeError** exception.
3. Let *extensible* be the result of calling the [[IsExtensible]] internal method of *O*.
4. If *extensible* is **false**, then throw a **TypeError** exception.
5. If Type(*pattern*) is Object and *pattern* has a [[RegExpMatcher]] internal data property, then
   a. If the value of *pattern's* [[RegExpMatcher]] internal data property is **undefined**, then throw a **TypeError** exception.
   b. If *flags* is not **undefined**, then throw a **TypeError** exception.
   c. Let *P* be the value of *pattern's* [[OriginalSource]] internal data property.
   d. Let *F* be the value of *pattern's* [[OriginalFlags]] internal data property.
6. Else,
   a. let *P* be *pattern*.
   b. let *F* be *flags*.
7. Return the result of the abstract operation RegExpInitialise with arguments *obj*, *P*, and *F* .

Deleted: ize

NOTE The **`compile`** method completely reinitialised the this object RegExp with a new pattern and flags. An implementaton 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.

Deleted: z

B.3 Other Additional Features

B.3.1 __proto__ Property Names in Object Initialisers

In 11.1.5 the Property Definition Evaluation algorithm for the production *PropertyDefinition* **:** *PropertyName* **:** *AssignmentExpression* is replaced with the following:

Deleted: 11.1.5
Deleted: 11.1.5

PropertyDefinition **:** *PropertyName* **:** *AssignmentExpression*

1. Let *propName* be the result of evaluating *PropertyName*.
2. ReturnIfAbrupt(*propName*).
3. Let *exprValue* be the result of evaluating *AssignmentExpression*.
4. Let *propValue* be GetValue(*exprValue*).
5. ReturnIfAbrupt(*propValue*).
6. If *propName* is the string value "**`__proto__`**", then
   a. If Type(*v*) is neither Object or Null, then throw a **TypeError** exception.
   b. Return the result of calling the [[SetInheritance]] internal method of *object* with argument *propValue*.
7. Let *desc* be the Property Descriptor{[[Value]]: *propValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
8. Return the result of DefinePropertyOrThrow(*object*, *propName desc*).

B.3.1 Web Legacy Compatibility for Block-Level Function Declarations

Prior to the Sixth Edition, the ECMAScript specification did not define the occurrence of a *FunctionDeclaration* as an element of a *Block* statement's *StatementList*. However, support for that form of *FunctionDeclaration* was an allowable extension and most browser-hosted ECMAScript implementations permitted them. However, the semantics of such declarations differ among those implementations. Because of these semantic differences, existing web ECMAScript code that uses *Block* level function declarations is only portable among browser implementation if the usage only depends upon the semantic intersection of all of the browser implementations for such declarations. The following are the use cases that fall within that intersection semantics:

1. A function is declared and only referenced within a single block
   - A function declaration with the name *f* is declared exactly once within the function code of an enclosing function *g* and that declaration is nested within a *Block*.
   - No other declaration of *f* that is not a **`var`** declaration occurs within the function code of *g*
   - All references to *f* occur within the *StatementList* of the *Block* containing the declaration of *f*.
2. A function is declared and possibly used within a single *Block* but also referenced by an inner function definition that is not contained within that same *Block*.
   - A function declaration with the name *f* is declared exactly once within the function code of an enclosing function *g* and that declaration is nested within a *Block*.
   - No other declaration of *f* that is not a **`var`** declaration occurs within the function code of *g*
   - References to *f* may occur within the *StatementList* of the *Block* containing the declaration of *f*.
   - References to *f* occur within the function code of *g* that lexically follows the *Block* containing the declaration of *f*.
3. A function is declared and possibly used within a single block but also referenced within subsequent blocks.
   - A function declaration with the name *f* is declared exactly once within the function code of an enclosing function *g* and that declaration is nested within a *Block*.
   - No other declaration of *f* that is not a **`var`** declaration occurs within the function code of *g*
   - References to *f* may occur within the *StatementList* of the *Block* containing the declaration of *f*.
   - References to *f* occur within another function *h* that is nested within *g* and no other declaration of *f* shadows the references to *f* from within *h*.
   - All invocations of *h* occur after the declaration of *f* has been evaluated.

The first use case is interoperable with the inclusion of *Block* level function declarations in the sixth edition. Any pre-existing ECMAScript code that employees that use case will operate using the Block level function declarations semantics defined by clauses 10 and 13 of this specification.

Sixth edition interoperability for the second and third use cases requires the following extensions to the clauses 10 and 13 semantics. These extensions are applied to a non-strict mode functions *g* if the above pre-conditions of use cases 2 and 3 above exist at the time of static semantic analysis of *g*. However, the last pre-

Deleted: PropName of

Deleted: <#>**The __proto__ pseudo property.¶**
<#>**Object.prototype.__proto__¶**
The initial value of the **`__proto__`** property o
the Object prototype object is a data property
whose initial value is **null**. This property initia
has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }
Manipulations of this property as tracked by t
Boolean valued primordial internal variable
UnderscoreProtoEnabled. The default initial val
of UnderscoreProtoEnabled is **true** only if this
property is initially present on the primordial
Object prototype object.¶
NOTE Any modification of this property or its
attributes causes UnderscoreProtoEnabled to b
set to **false**.¶
<#> **Changes To Internal Methods__¶**
The definition of the [[GetP]] internal method
given in 8.12.3 is replaced with the following:
<#>If *P* is the string value "**`__proto__`**" and
UnderscoreProtoEnabled is **true**, then ¶
<#>Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.¶
<#>If *desc* is not **undefined** and was created step 1.a to describe the property defined in B.3.1.1 then,¶
<#>Return the value of the [[Prototype]] internal data property of *O*.¶
<#>Continue by executing the steps of 8.12.3 starting with step 1.¶
¶
The definition of the [[Put]] internal method
given in 8.12.5 is replaced with the following:
<#>If *P* is the string value "**`__proto__`**" and
UnderscoreProtoEnabled is **true** and *O* is not the
standard built-in Object prototype object, then ¶
<#>Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.¶
<#>If *desc* is not **undefined** and was created step 1.a to describe the property defined in B.3.1.1 then,¶
<#>If the type of *V* is neither Object or N
return¶
<#>Set the value of the [[Prototype]] inter
data property of *O* to *V*.¶
<#>Return.¶
<#>Continue by executing the steps of 8.12.5 starting with step 1.¶
The definition of the [[Delete]] internal metho
given in 8.12.7 is replaced with the following:
<#>If UnderscoreProtoEnabled is **true** and *P* is
the string value "**`__proto__`**" and *O* is the
standard built-in Object prototype object, then ¶
<#>Set UnderscoreProtoEnabled to **false**.¶
<#>Continue by executing the steps of 8.12.7 starting with step 1.¶
¶
The definition of the [[DefineOwnProperty]]
internal method given in 8.12.9 is replaced w
the following:¶
<#>If UnderscoreProtoEnabled is **true** and *P* is
the string value "**`__proto__`**" and *O* is the
standard built-in Object prototype object, then ¶
<#>If any attribute contained in *Desc* is not
present or has a different value from the
corresponding attribute in { [[Writable]]: **true**

condition of use case 3 is not included in this determination and the determination is not applied to any function declaration that is nested within syntactic constructs that are specified in the Fifth edition of this specification.

1. Let *B* be environment record for the construct within *g* that introduces a new environment contour and which most closely encloses the declaration of *f*, all function code references to *f*, and the definitions of all nested functions that contain unshadowed references to *f*. This syntactic construct may be the definition of *g* itself, in which case *B* is the function environment record for *g*.
2. As part of the instantiation of *B*, its CreateMutableBinding concrete method is called with arguments "f" (the string name of the function) and **false**. This creates an unitialised binding for the name *f*. Any reference that resolves to that binding prior to step 3 below will throw a **ReferenceError** exception.
3. When the InitialiseBinding concrete method is used to initialise the binding for the function declaration *f* also invoke InitialiseBind on *B* using the same arguments.

Deleted: ize

If an ECMAScript implication has a mechanism that produces diagnostic warning messages, a warning should be produced for each function *g* for which the above steps are performed.

Annex C
(informative)

The Strict Mode of ECMAScript

The strict mode restriction and exceptions

- The identifiers "**implements**", "**interface**", "**let**", "**package**", "**private**", "**protected**", "**public**", "**static**", and "**yield**" are classified as *FutureReservedWord* tokens within strict mode code. (7.6.1.2).
- A conforming implementation, when processing strict mode code, may not extend the syntax of *NumericLiteral* (7.8.3) to include *OctalIntegerLiteral* as described in B.1.1.
- A conforming implementation, when processing strict mode code (see 10.1.1), may not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.2.
- Assignment to an undeclared identifier or otherwise unresolvable reference does not create a property in the global object. When a simple assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable Reference. If it does a **ReferenceError** exception is thrown (8.9.2). The *LeftHandSide* also may not be a reference to a data property with the attribute value {[[Writable]]:**false**}, to an accessor property with the attribute value {[[Set]]:**undefined**}, nor to a non-existent property of an object whose [[Extensible]] internal data property has the value **false**. In these cases a **TypeError** exception is thrown (11.13.1).
- The identifier **eval** or **arguments** may not appear as the *LeftHandSideExpression* of an Assignment operator (11.13) or of a *PostfixExpression* (11.3) or as the *UnaryExpression* operated upon by a Prefix Increment (11.4.4) or a Prefix Decrement (11.4.5) operator.
- Arguments objects for strict mode functions define non-configurable accessor properties named "**caller**" and "**callee**" which throw a **TypeError** exception on access (10.6).
- Arguments objects for strict mode functions do not dynamically share their array indexed property values with the corresponding formal parameter bindings of their functions. (10.6).
- For strict mode functions, if an arguments object is created the binding of the local identifier **arguments** to the arguments object is immutable and hence may not be the target of an assignment expression. (10.5).
- It is a **SyntaxError** if strict mode code contains an *ObjectLiteral* with more than one definition of any data property (11.1.5).
- It is a **SyntaxError** if the *Identifier* **"eval"** or the *Identifier* **"arguments"** occurs as the *Identifier* in a *PropertySetParameterList* of a *PropertyDefinition* that is contained in strict code or if its *FunctionBody* is strict code (11.1.5).
- Strict mode eval code cannot instantiate variables or functions in the variable environment of the caller to eval. Instead, a new variable environment is created and that environment is used for declaration binding instantiation for the eval code (10.4.2).
- If **this** is evaluated within strict mode code, then the **this** value is not coerced to an object. A **this** value of **null** or **undefined** is not converted to the global object and primitive values are not converted to wrapper objects. The **this** value passed via a function call (including calls made using **Function.prototype.apply** and **Function.prototype.call)** do not coerce the passed this value to an object (10.4.3, 11.1.1, 15.3.3.3, 15.3.3.4).
- 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(11.4.1).

Deleted: 7

Deleted: *PropertyAssignment*

Deleted: 15.3.4

Deleted: 15.3.4

- When a **`delete`** operator occurs within strict mode code, a **TypeError** is thrown if the property to be deleted has the attribute { [[Configurable]]:**false** } (11.4.1).
- It is a **SyntaxError** if a *VariableDeclaration* or *VariableDeclarationNoIn* occurs within strict code and its *Identifier* is **`eval`** or **`arguments`** (12.2.1).
- Strict mode code may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context is an **SyntaxError** (12.10).
- It is a **SyntaxError** if a *TryStatement* with a *Catch* occurs within strict code and the *Identifier* of the *Catch* production is **`eval`** or **`arguments`** (12.14.1)
- It is a **SyntaxError** if the identifier **`eval`** or **`arguments`** appears within the *FormalParameters* of a strict mode *FunctionDeclaration* or *FunctionExpression* (13.1)
- A strict mode function may not have two or more formal parameters that have the same name. An attempt to create such a function using a *FunctionDeclaration*, *FunctionExpression*, or **`Function`** constructor is a **SyntaxError** (13.1, 15.3.1).
- An implementation may not extend, beyond that defined in this specification, the meanings within strict mode functions of properties named **`caller`** or **`arguments`** of function instances. ECMAScript code may not create or modify properties with these names on function objects that correspond to strict mode functions (8.3.15.6, 10.6, 15.3.3.5.3).
- It is a **SyntaxError** to use within strict mode code the identifiers **`eval`** or **`arguments`** as the *Identifier* of a *FunctionDeclaration* or *FunctionExpression* or as a formal parameter name (13.1). Attempting to dynamically define such a strict mode function using the **`Function`** constructor (15.3.1) will throw a **SyntaxError** exception.

Deleted: a

Deleted: *FormalParameterList*

Deleted: 2

Deleted: 13.26, 15.3.4

Deleted: 2

Annex D
(informative)
Corrections and Clarifications with Possible Compatibility Impact

Deleted: in the 5^{th} Edition

Deleted: 3^{rd} Edition

In Edition 6

Deleted: the 6th

15.9.1.14: Previous editions permitted the TimeClip abstract operation to return either +0 or –0 as the representation of a 0 time value. The 6^{th} Edition specifies that +0 always returned. This means that for the 6^{th} Edition the time value of a Date object is never observably –0 and methods that return time values never return –0.

15.9.1.15: If a time zone offset is not present, the local time zone is used. Edition 5.1 incorrectly stated that a missing time zone should be interpreted as "z".

15.9.5.2: Previous editions did not specify the value returned by Date.prototype.toString when this time value is NaN. The 6^{th} Edition specifies the result to be the String value is **`"Invalid Date"`**

Deleted: species

In Edition 5.1

Deleted: 5.1

7.8.4: CV definitions added for *DoubleStringCharacter* **::** *LineContinuation* and *SingleStringCharacter* **::** *LineContinuation*.

10.2.1.1.3: The argument S is not ignored. It controls whether an exception is thrown when attempting to set an immutable binding.

10.2.1.2.2: In algorithm step 5, **true** is passed as the last argument to [[DefineOwnProperty]].

10.5: Former algorithm step 5.e is now 5.f and a new step 5.e was added to restore compatibility with 3^{rd} Edition when redefining global functions.

11.5.3: In the final bullet item, use of IEEE 754 round-to-nearest mode is specified.

12.6.3: Missing ToBoolean restored in step 3.a.ii of both algorithms.

12.6.4: Additional final sentences in each of the last two paragraphs clarify certain property enumeration requirements.

12.7, 12.8, 12.9: BNF modified to clarify that a **`continue`** or **`break`** statement without an *Identifier* or a **`return`** statement without an *Expression* may have a *LineTerminator* before the semi-colon.

12.14: Step 3 of algorithm 1 and step 2.a of algorithm 3 are corrected such that the value field of *B* is passed as a parameter rather than *B* itself.

15.1.2.2: In step 2 of algorithm, clarify that *S* may be the empty string.

15.1.2.3: In step 2 of algorithm clarify that *trimmedString* may be the empty string.

15.1.3: Added notes clarifying that ECMAScript's URI syntax is based upon RFC 2396 and not the newer RFC 3986. In the algorithm for Decode, a step was removed that immediately preceded the current step 4.d.vii.10.a because it tested for a condition that cannot occur.

15.2.3.7: Corrected use of variable *P* in steps 5 and 6 of algorithm.

15.2.4.2: Edition 5 handling of **undefined** and **null** as **this** value caused existing code to fail. Specification modified to maintain compatibility with such code. New steps 1 and 2 added to the algorithm.

15.3.3.3: Steps 5 and 7 of Edition 5 algorithm have been deleted because they imposed requirements upon the *argArray* argument that are inconsistent with other uses of generic array-like objects.

Deleted: 15.3.4

15.4.3.12: In step 9.a, incorrect reference to *relativeStart* was replaced with a reference to *actualStart*.

Deleted: 15.4.4

15.4.3.15: Clarified that the default value for *fromIndex* is the length minus 1 of the array.

Deleted: 15.4.4

15.4.3.18: In step 10 (corresponding to step 8 in 5.1) of the algorithm, **undefined** is now the specified return value.

Deleted: 15.4.4

Deleted: 9

15.4.3.22: In step 11.d.iii (corresponding to 9.c.ii in 5.1) the first argument to the [[Call]] internal method has been changed to **undefined** for consistency with the definition of `Array.prototype.reduce`.

Deleted: 15.4.4

Deleted: 9

Deleted: c

15.4.5.1: In Algorithm steps 3.l.ii and 3.l.iii the variable name was inverted resulting in an incorrectly inverted test.

15.5.4.9: Normative requirement concerning canonically equivalent strings deleted from paragraph following algorithm because it is listed as a recommendation in NOTE 2.

15.5.4.14: In `split` algorithm step 11.a and 13.a, the positional order of the arguments to *SplitMatch* was corrected to match the actual parameter signature of *SplitMatch*. In step 13.a.iii.7.d, *lengthA* replaces *A*.`length`.

15.5.5.2: In first paragraph, removed the implication that the individual character property access had “array index” semantics. Modified algorithm steps 3 and 5 such that they do not enforce “array index” requirement.

15.9.1.15: Specified legal value ranges for fields that lacked them. Eliminated “time-only” formats. Specified default values for all optional fields.

15.10.2.2: The step numbers of the algorithm for the internal closure produced by step 2 were incorrectly numbered in a manner that implied that they were steps of the outer algorithm.

15.10.2.6: In the abstract operation *IsWordChar* the first character in the list in step 3 is “`a`” rather than “`A`”.

15.10.2.8: In the algorithm for the closure returned by the abstract operation *CharacterSetMatcher*, the variable defined by step 3 and passed as an argument in step 4 was renamed to *ch* in order to avoid a name conflict with a formal parameter of the closure.

15.10.6.2: Step 9.e was deleted because It performed an extra increment of *i*.

15.11.1.1: Removed requirement that the `message` own property is set to the empty String when the *message* argument is **undefined**.

15.11.1.2: Removed requirement that the `message` own property is set to the empty String when the *message* argument is **undefined**.

15.11.4.4: Steps 6-10 modified/added to correctly deal with missing or empty `message` property value.

15.11.1.2: Removed requirement that the `message` own property is set to the empty String when the *message* argument is **undefined**.

15.12.3: In step 10.b.iii of the *JA* abstract operation, the last element of the concatenation is “`]`”.

Deleted: internal

B.2.1: Added to NOTE that the encoding is based upon RFC 1738 rather than the newer RFC 3986.

Annex C: An item was added corresponding to 7.6.12 regarding *FutureReservedWords* in strict mode.

In Edition 5

Throughout: In the Edition 3 specification the meaning of phrases such as "as if by the expression **`new Array()`**" are subject to misinterpretation. In the Edition 5 specification text for all internal references and invocations of standard built-in objects and methods has been clarified by making it explicit that the intent is that the actual built-in object is to be used rather than the current dynamically resolved value of the correspondingly identifier binding.

11.8.1: ECMAScript generally uses a left to right evaluation order, however the Edition 3 specification language for the > and <= operators resulted in a partial right to left order. The specification has been corrected for these operators such that it now specifies a full left to right evaluation order. However, this change of order is potentially observable if side-effects occur during the evaluation process.

11.1.4: Edition 5 clarifies the fact that a trailing comma at the end of an *ArrayInitialiser* does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

11.2.3: Edition 5 reverses the order of steps 2 and 3 of the algorithm. The original order as specified in Editions 1 through 3 was incorrectly specified such that side-effects of evaluating *Arguments* could affect the result of evaluating *MemberExpression*.

12.4: In Edition 3, an object is created, as if by **`new Object()`** to serve as the scope for resolving the name of the exception parameter passed to a **`catch`** clause of a **`try`** statement. If the actual exception object is a function and it is called from within the **`catch`** clause, the scope object will be passed as the **this** value of the call. The body of the function can then define new properties on its **this** value and those property names become visible identifiers bindings within the scope of the **catch** clause after the function returns. In Edition 5, when an exception parameter is called as a function, **undefined** is passed as the **this** value.

13: In Edition 3, the algorithm for the production *FunctionExpression* with an *Identifier* adds an object created as if by **`new Object()`** to the scope chain to serve as a scope for looking up the name of the function. The identifier resolution rules (10.1.4 in Edition 3) when applied to such an object will, if necessary, follow the object's prototype chain when attempting to resolve an identifier. This means all the properties of Object.prototype are visible as identifiers within that scope. In practice most implementations of Edition 3 have not implemented this semantics. Edition 5 changes the specified semantics by using a Declarative Environment Record to bind the name of the function.

14: In Edition 3, the algorithm for the production *SourceElements* **:** *SourceElements SourceElement* did not correctly propagate statement result values in the same manner as *Block*. This could result in the **`eval`** function producing an incorrect result when evaluating a *Program* text. In practice most implementations of Edition 3 have implemented the correct propagation rather than what was specified in Edition 5.

15.10.6: RegExp.prototype is now a RegExp object rather than an instance of Object. The value of its [[Class]] internal data property which is observable using Object.prototype.toString is now "RegExp" rather than "Object".

Deleted: the
Deleted: 5th
Deleted: named property
Deleted: 2
Deleted: , 11.8.3, 11.8.5

Annex E
(informative)

Additions and Changes that Introduce Incompatibilities with Prior Editions

Deleted: in the 5th Edition

Deleted: the 3rd

16.2 In the 6th Edition

11.2.3: In Edition 6, Function calls are not allowed to return a Reference value.

12.6: In Edition 6, a terminating semi-colon is no longer required at the end of a do-while statement.

12.14: In Edition 6, it is an early error for a *Catch* clause to contained a **`var`** declaration for the same *Identifier* that appears as the *Catch* clause parameter. In previous editions, such a variable declaration would be instantiated in the enclosing variable environment but the declaration's *Initialiser* value would be assigned to the *Catch* parameter.

Deleted:

Deleted: *ize*

13.3 In Edition 6, the function objects that are created as the values of the [[Get]] or [[Set]] attribute of accessor properties in an *ObjectLiteral* are not constructor functions. In Edition 5, they were constructors.

Deleted: t

Deleted: *I*

15.2.3.2: In Edition 6, if the argument to **`Object.getPrototypeOf`** is not an object an attempt is make to coerce the argument using ToObject. If the coerecion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.3: In Edition 6, if the argument to **`Object.getOwnPropertyDescriptor`** is not an object an attempt is make to coerce the argument using ToObject. If the coerecion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.4: In Edition 6, if the argument to **`Object.getOwnPropertyNames`** is not an object an attempt is make to coerce the argument using ToObject. If the coerecion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.5 and 15.2.3.7: In Edition 6, all property additions and changes are processed, even if one of them throws an exception. If an exception occurs during such processing, the first such exception is thrown after all propertie are processed. In Edition 5, processing of property additions and changes immediately terminated when the first exception occurred.

Deleted: acc

15.2.3.8: In Edition 6, if the argument to **`Object.seal`** is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.9: In Edition 6, if the argument to **`Object.freeze`** is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.10: In Edition 6, if the argument to **`Object.preventExtensions`** is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.11: In Edition 6, if the argument to **`Object.isSealed`** is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.12: In Edition 6, if the argument to `Object.isFrozen` is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.13: In Edition 6, if the argument to `Object.isExtensible` is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.2.3.14: In Edition 6, if the argument to `Object.keys` is not an object an attempt is make to coerce the argument using ToObject. If the coerecion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a **TypeError** to be thrown.

15.5.4 In Edition 6, the Array prototype object is not a Arrayn instance. In previous editions it was an Array instance with a length property whose value was +0.

15.5.4 In Edition 6, the String prototype object is not a String instance. In previous editions it was a String instance whose String value was the empty string.

15.6.4 In Edition 6, the Boolean prototype object is not a Boolean instance. In previous editions it was a Boolean instance whose Boolean value was **false**.

15.7.4 In Edition 6, the Number prototype object is not a Number instance. In previous editions it was a Number instance whose number value was +0.

15.9.5 In Edition 6, the Date prototype object is not a Date instance. In previous editions it was a Date instance whose TimeValue was NaN.

15.10.6 In Edition 6, the RegExp prototype object is not a RegExp instance. In previous editions it was a RegExp instance whose pattern is the empty string.

15.10.7 In Edition 6, `source`, `global`, `ignoreCase`, and `multiline` are accessor properties defined on the RegExp prototype object. In previous editions they were data properties defined on RegExp instances.

Deleted: number

Deleted: number

Deleted: Date

16.3 In the 5th Edition

7.1: Unicode format control characters are no longer stripped from ECMAScript source text before processing. In Edition 5, if such a character appears in a *StringLiteral* or *RegularExpressionLiteral* the character will be incorporated into the literal where in Edition 3 the character would not be incorporated into the literal.

7.2: Unicode character <BOM> is now treated as whitespace and its presence in the middle of what appears to be an identifier could result in a syntax error which would not have occurred in Edition 3

7.3: Line terminator characters that are preceded by an escape sequence are now allowed within a string literal token. In Edition 3 a syntax error would have been produced.

7.8.5: Regular expression literals now return a unique object each time the literal is evaluated. This change is detectable by any programs that test the object identity of such literal values or that are sensitive to the shared side effects.

7.8.5: Edition 5 requires early reporting of any possible RegExp constructor errors that would be produced when converting a *RegularExpressionLiteral* to a RegExp object. Prior to Edition 5 implementations were permitted to defer the reporting of such errors until the actual execution time creation of the object.

7.8.5: In Edition 5 unescaped "/" characters may appear as a *CharacterClass* in a regular expression literal. In Edition 3 such a character would have been interpreted as the final character of the literal.

10.4.2: In Edition 5, indirect calls to the `eval` function use the global environment as both the variable environment and lexical environment for the eval code. In Edition 3, the variable and lexical environments of the caller of an indirect `eval` was used as the environments for the eval code.

15.4.3: In Edition 5 all methods of **`Array.prototype`** are intentionally generic. In Edition 3 **`toString`** and **`toLocaleString`** were not generic and would throw a **`TypeError`** exception if applied to objects that were not instances of Array.

Deleted: 15.4.4

10.6: In Edition 5 the array indexed properties of argument objects that correspond to actual formal parameters are enumerable. In Edition 3, such properties were not enumerable.

10.6: In Edition 5 the value of the [[Class]] internal data property of an arguments object is **`"Arguments"`**. In Edition 3, it was **`"Object"`**. This is observable if **`toString`** is called as a method of an arguments object.

12.6.4: for-in statements no longer throw a **TypeError** if the **`in`** expression evaluates to **null** or **undefined**. Instead, the statement behaves as if the value of the expression was an object with no enumerable properties.

15: In Edition 5, the following new properties are defined on built-in objects that exist in Edition 3: **`Object.getPrototypeOf`**, **`Object.getOwnPropertyDescriptor`**, **`Object.getOwnPropertyNames`**, **`Object.create`**, **`Object.defineProperty`**, **`Object.defineProperties`**, **`Object.seal`**, **`Object.freeze`**, **`Object.preventExtensions`**, **`Object.isSealed`**, **`Object.isFrozen`**, **`Object.isExtensible`**, **`Object.keys`**, **`Function.prototype.bind`**, **`Array.prototype.indexOf`**, **`Array.prototype.lastIndexOf`**, **`Array.prototype.every`**, **`Array.prototype.some`**, **`Array.prototype.forEach`**, **`Array.prototype.map`**, **`Array.prototype.filter`**, **`Array.prototype.reduce`**, **`Array.prototype.reduceRight`**, **`String.prototype.trim`**, **`Date.now`**, **`Date.prototype.toISOString`**, **`Date.prototype.toJSON`**.

15: Implementations are now required to ignore extra arguments to standard built-in methods unless otherwise explicitly specified. In Edition 3 the handling of extra arguments was unspecified and implementations were explicitly allowed to throw a **TypeError** exception.

15.1.1: The value properties **NaN**, **Infinity**, and **undefined** of the Global Object have been changed to be read-only properties.

15.1.2.1. Implementations are no longer permitted to restrict the use of eval in ways that are not a direct call. In addition, any invocation of eval that is not a direct call uses the global environment as its variable environment rather than the caller's variable environment.

15.1.2.2: The specification of the function **`parseInt`** no longer allows implementations to treat Strings beginning with a **`0`** character as octal values.

15.3.3.3: In Edition 3, a **TypeError** is thrown if the second argument passed to **`Function.prototype.apply`** is neither an array object nor an arguments object. In Edition 5, the second argument may be any kind of generic array-like object that has a valid **`length`** property.

Deleted: 15.3.4

15.3.3.3, 15.3.3.4: In Edition 3 passing **undefined** or **null** as the first argument to either **`Function.prototype.apply`** or **`Function.prototype.call`** causes the global object to be passed to the indirectly invoked target function as the **this** value. If the first argument is a primitive value the result of calling ToObject on the primitive value is passed as the **this** value. In Edition 5, these transformations are not performed and the actual first argument value is passed as the **this** value. This difference will normally be unobservable to existing ECMAScript Edition 3 code because a corresponding transformation takes place upon activation of the target function. However, depending upon the implementation, this difference may be observable by host object functions called using **`apply`** or **`call`**. In addition, invoking a standard built-in function in this manner with **null** or **undefined** passed as the **this** value will in many cases cause behaviour in Edition 5 implementations that differ from Edition 3 behaviour. In particular, in Edition 5 built-in functions that are specified to actually use the passed **this** value as an object typically throw a **TypeError** exception if passed **null** or **undefined** as the **this** value.

Deleted: 15.3.4

Deleted: 15.3.4

15.3.4.2: In Edition 5, the **`prototype`** property of Function instances is not enumerable. In Edition 3, this property was enumerable.

Deleted: 15.3.5

15.5.5.2: In Edition 5, the individual characters of a String object's [[StringData]] may be accessed as array indexed properties of the String object. These properties are non-writable and non-configurable and shadow any inherited properties with the same names. In Edition 3, these properties did not exist and ECMAScript code could dynamically add and remove writable properties with such names and could access inherited properties with such names.

Deleted: PrimitiveValue

15.9.4.2: **`Date.parse`** is now required to first attempt to parse its argument as an ISO format string. Programs that use this format but depended upon implementation specific behaviour (including failure) may behave differently.

15.10.2.12: In Edition 5, **`\s`** now additionally matches <BOM>.

15.10.4.1: In Edition 3, the exact form of the String value of the **`source`** property of an object created by the **`RegExp`** constructor is implementation defined. In Edition 5, the String must conform to certain specified requirements and hence may be different from that produced by an Edition 3 implementation.

15.10.6.4: In Edition 3, the result of **`RegExp.prototype.toString`** need not be derived from the value of the RegExp object's **`source`** property. In Edition 5 the result must be derived from the **`source`** property in a specified manner and hence may be different from the result produced by an Edition 3 implementation.

15.11.2.1, 15.11.4.3: In Edition 5, if an initial value for the **`message`** property of an Error object is not specified via the **`Error`** constructor the initial value of the property is the empty String. In Edition 3, such an initial value is implementation defined.

15.11.4.4: In Edition 3, the result of **`Error.prototype.toString`** is implementation defined. In Edition 5, the result is fully specified and hence may differ from some Edition 3 implementations.

15.12: In Edition 5, the name **`JSON`** is defined in the global environment. In Edition 3, testing for the presence of that name will show it to be **undefined** unless it is defined by the program or implementation.

Annex F
(informative)

Static Semantic Rule Cross Reference

Routine Name	Purpose	Definitions	Uses
BoundNames	Produces a list of the Identifiers bound by a production. Does not include Identifiers that are bound within inner environments associated with the production.	12.2.1, 12.2.2, 12.2.4, 12.6.4, 13.1, 13.2, 13.5	
ConstructorMethod	From a *ClassBody* return the first *ClassElement* whose PropName is **"constructor"**. Returns empty if the *ClassBody* does not contain one.	13.5	
Contains	Determine if a grammar production either directly or indirectly includes a grammar symbol.	5.3, 13.1, 13.2, 13.5	
CoveredFormalsList	Reparse a covered *Expression* using *FormalsList* as the goal symbol.	13.2	
CV	Determines the "character value" of a component of a *StringLiteral*.	7.8.4	
Elision Width	Determine the number of commas in an *Elision*.	11.1.4.1	
ExpectedArgumentCount	Determine the "length" of an argument list for the purpose of initializing the "length" property of a function object.	13.1, 13.2, 13.3	
HasInitialiser	Determines whether the production contains an *Initialiser* production.	12.2.4, 13.1	
IsConstantDeclaration	Determines whether the production introduces a immutable environment record binding	12.2, 13.1, 13.5	
IsInvalidAssignmentPattern	Determines if a *LeftHandSideExpression* is a valid assignment target. Primarily for dealing with destructuring assignment targets.	11.2	
LexicalDeclarations	Return a List containing the components of a production that are processed as lexical declarations	12.1, 12.11, 12.5	
LexicallyDeclaredNames	Returns a list of the lexically scoped identifiers declared by a production.	12.1, 13.1, 13.2, 13.5	
PrototypeMethodDefinitions	Return a list of the non-static *MethodDefinition* productions that are part of a *ClassElementList*.	13.5	
MV	Determines the "mathematical value" of a numeric	7.8.3	

Deleted: Technically Significant Corrections and Clarifications in the 5.1 Edition

Deleted: s

	lirteral or component of a numeric literal.		
PropName	Determines the string value of the property name referenced by a production.	11.1.5.1, 13.3, 13.5	
PropNameList	Returns a List of the string values of the property names referenced by a production. The list reflects the order of the references in the source text. The list may contain duplicate elements.	11.5.1, 13.5	
PrototypeMethodDefinitions	Return a list of the non-static *MethodDefinition* productions that are part of a *ClassElementList*.	13.5	
ReferencesSuper	Determine if a *MethodDefinition* contains any references to the *ReservedWord* **`super`**.	13.3	
SpecialMethod	Determine if a *MethodDefinition* defines a generator method or an accessor property.	13.3	
StaticMethodDefinitions	Return a list of the static *MethodDefinition* productions that are part of a *ClassElementList*.	13.5	
SV	Determines the "string value" of a *StringLiteral* or component of a *StringLiteral*.	7.8.4	
VarDeclaredNames	Returns a list of the local top-level scoped identifiers declared by a production. These are identifier that are scoped as if by a var statement.	12.1, 12.5, 12.6.1, 12.6.2, 12.6.3, 12.6.4, 12.12, 13.1, 13.5	

Formatted: Heading 1

Scrap Heap

A place to temporarily hand on to stuff that's been deleted

MemberExpression **:**
  MemberExpression **<|** *TriangleLiteral*

TriangleLiteral **:**
  SealedArrayLiteral
  SealedObjectLiteral
  FunctionExpression
  ArrowFunction
  ValueLiteral

CallExpression **:**
  CallExpression **<|** *TriangleLiteral*

Formatted: Normal

16.3.1.1 15.2.3.15 Object.isObject (O)

When the **isObject** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is Object return **true**.
2. Return **false**.

15.5.4.25 String.prototype.toArray()

The following steps are taken:

1. Let *O* be CheckObjectCoercible(**this** value).
2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *len* be the number of characters in *S*.
5. Let *array* be the result of the abstract operation ArrayCreate with argument *len*.
6. Let *n* be 0
7. Repeat, while *n* < *len*:
   a. Let *c* be the character at position *n* in *S*.
   b. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*n*), the PropertyDescriptor {[[Value]]: *c*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
   c. Increment *n* by 1.
8. Return *array*.

Deleted: ReturnIfAbrupt(

Deleted:)

Deleted: the result of calling

Deleted: , giving it the **this** value as its argument

Deleted: (15.4)

The **length** property of the **toArray** method is **0**.

NOTE 1 Returns an Array object with elements corresponding to the characters of this object (converted to a String).

NOTE 2 The **toArray** 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.

Static Semantics: TopLevelLexicallyDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be TopLevelLexicallyDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the TopLevelLexicallyDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *StatementListItem*

1. Return a new empty List.

StatementListItem **:** *Declaration*

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

16.3.2 Preliminary work on Irrefutable Destructuring Binding Patterns

Syntax

BindingPattern **:**
    Irrefutable$_{opt}$ *ObjectBindingPattern*
    Irrefutable$_{opt}$ *ArrayBindingPattern*

Irrefutable **:**
    ?

ObjectBindingPattern **:**
    { }
    { *BindingPropertyList* **}**
    { *BindingPropertyList* **, }**

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

BindingPropertyList **:**
    Irrefutable$_{opt}$ *BindingProperty*
    BindingPropertyList **,** *Irrefutable*$_{opt}$ *BindingProperty*

BindingElementList **:**
    Elision$_{opt}$ *BindingElement*
    BindingElementList **,** *Elision*$_{opt}$ *BindingElement*

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

Comment [AWB16236]: Note that this ma be a computed property name

BindingElement **:**
    SingleNameBinding
    BindingPattern Initialiser$_{opt}$

SingleNameBinding **:**
    BindingIdentifier Initialiser$_{opt}$

BindingRestElement **:**
    . . . *BindingIdentifier*

16.3.2.1 Static Semantics

Static Semantics: Early Errors

BindingPattern **:** *Irrefutable*$_{opt}$ *ObjectBindingPattern*

- It is a Syntax Error if the BoundNames of *ObjectBindingPattern* contains the string "**`eval`**" or the string "**`arguments`**".

BindingPattern **:** *Irrefutable*$_{opt}$ *ArrayBindingPattern*

- It is a Syntax Error if the BoundNames of *ArrayBindingPattern* contains the string "**`eval`**" or the string "**`arguments`**".

Static Semantics: BoundNames

BindingPattern **:** *Irrefutable*$_{opt}$ *ObjectBindingPattern*

1. Return the BoundNames of *ObjectBindingPattern*.

BindingPattern **:** *Irrefutable*$_{opt}$ *ArrayBindingPattern*

2. Return the BoundNames of *ArrayBindingPattern*.

ObjectBindingPattern **: `{ }`**

3. Return an empty List.

ArrayBindingPattern **: `[`** *Elision*$_{opt}$ **`]`**

2. Return an empty List.

ArrayBindingPattern **: `[`** *Elision*$_{opt}$ *BindingRestElement* **`]`**

2. Return the BoundNames of *BindingRestElement*.

ArrayBindingPattern **: `[`** *BindingElementList* **`,`** *Elision*$_{opt}$ **`]`**

2. Return the BoundNames of *BindingElementList*.

ArrayBindingPattern **: `[`** *BindingElementList* **`,`** *Elision*$_{opt}$ *BindingRestElement* **`]`**

4. Let *names* be BoundNames of *BindingElementList*.
5. Append to *names* the elements of BoundNames of *BindingRestElement*.
6. Return *names*.

BindingPropertyList **:** *Irrefutable*$_{opt}$ *BindingProperty*

1. Return the BoundNames of *BindingProperty*.

BindingPropertyList **:** *BindingPropertyList* **`,`** *Irrefutable*$_{opt}$ *BindingProperty*

4. Let *names* be BoundNames of *BindingPropertyList*.
5. Append to *names* the elements of BoundNames of *BindingProperty*.
6. Return *names*.

BindingElementList **:** *Elision*$_{opt}$ *BindingElement*

2. Return BoundNames of *BindingElement*.

BindingElementList **:** *BindingElementList* **,** *Elision*$_{opt}$ *BindingElement*

4. Let *names* be BoundNames of *BindingElementList*.
5. Append to *names* the elements of BoundNames of *BindingElement*.
6. Return *names*.

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

2. Return the BoundNames of *BindingElement*.

SingleNameBinding **:** *BindingIdentifier Initialiser*$_{opt}$

2. Return the BoundNames of *BindingIdentifier*.

BindingElement **:** *BindingPattern Initialiser*$_{opt}$

2. Return the BoundNames of *BindingPattern*.

16.3.3 8.3.10 [[Enumerate]] (includePrototype, onlyEnumerable)

When the [[Enumerate]] internal method of *O* is called with Boolean arguments *includePrototype* and *onlyEnumerable*, the following steps are taken:

1. Return an Iterator object (reference xxxx) whose next method iterates over all the keys of enumerable property keys of *O*. If *includePrototype* is **false**, then only own properties of *O* are included. If *onlyEnumerable* is **false**, then all properties that do not have private name keys are included. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.

Enumerated properties do not include properties whose property key is a private name. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name. The values of [[Enumerable]] attributes are not considered when determining if a property of a prototype object is shadowed by a previous object on the prototype chain.

The following is an informative algorithm that conforms to these rules

1. Let *obj* be *O*.
2. Let *proto* be the value of the [[Prototype]] internal data property of *O*.
3. If *includePrototype* is **false** or *proto* is the value **null**, then
    a. Let *propList* be a new empty List.
4. Else
    a. Let *propList* be the result of calling the [[Enumerate]] internal method of *proto* with arguments **true** and *onlyEnumerable*.
5. For each string *name* that is the property key of an own property of *O*
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *name*.
    b. If *name* is an element of *propList*, then remove *name* as an element of *propList*.
    c. If *onlyEnumerable* is **false** or *desc*.[[Enumerable]] is **true**, then add *name* as an element of *propList*.
6. Order the elements of *propList* in an implementation defined order.
7. Return *propList*.

Comment [AWB6237]: TODO

Comment [AWB6238]: TODO

Comment [AWB6239]: This is an experim to see if we can use this internal method to define Object.key, Object.getOwnPropertyNames, and perhaps some offther things

Deleted: <#>8.4.4 Symbol Exotic Obje
A *Symbol object* is an exotic object that may used as a property key. Symbol exotic object are unique in that they are always immutable and never observably reference any other object.¶
Exotic String objects have a single internal da properties named [[Private]] that is set when object is created and never modified. ¶
Exotic Symbol objects provide alternative definitions for all of the essential internal methods. ¶
<#>8.4.4.1 [[GetInheritance]] ( )¶
When the [[GetInheritance]] internal method an exotic Symbol object *O* is called the following steps are taken:¶
<#>Return **null**.¶
<#>8.4.4.2 [[SetInheritance]] (V)¶
When the [[SetInheritance]] internal method an exotic Symbol object *O* is called with argument *V* the following steps are taken:¶
<#>Assert: Either Type(*V*) is Object or Type(*V* Null.¶
<#>Return **false**.¶
<#>8.4.4.3 [[HasIntegrity]] (Level)¶
When the [[HasIntegrity]] internal method of a exotic Symbol object *O* is called the following steps are taken:¶
<#>Assert: *Level* is one of "**nonextensible** "**sealed**", or "**frozen**".¶
<#>Return **true**.¶
<#>8.4.4.4 [[SetIntegrity]] (Level)¶
When the [[SetIntegrity]] internal method of an exotic Symbol object an exotic Symbol object is called the following steps are taken:¶
<#>Assert: *Level* is one of "**nonextensible** "**sealed**", or "**frozen**".¶
<#>Return **true**.¶
<#>8.4.4.5 [[HasOwnProperty]] (P)¶
When the [[HasOwnProperty]] internal metho of an exotic Symbol object *O* is called with property key *P*, the following steps are taken:
<#>Return **false**.¶
<#>8.4.4.6 [[GetOwnProperty]] (P)¶
When the [[GetOwnProperty]] internal metho of an exotic Symbol object *O* is called with property key *P*, the following steps are taken:
<#>Return **undefined**.¶
<#>8.4.4.7 [[DefineOwnProperty]] (P, Desc)¶
When the [[DefineOwnProperty]] internal method of an exotic Symbol object *O* is called with property key *P* and property descriptor *D* the following steps are taken:¶
<#>Return **false**.¶

Comment [AWB6240]: TODO

Comment [AWB6241]: TODO: Finish this and turn it into iterator definition include a ne method

This follow version places function body declarations in scope of parameter initialisers

16.3.4 9.1.11 ToPositiveInteger

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

1. Let *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, return **+0**.
4. If *number* is +∞, or –∞, return *number*.
5. If *number* ≤ **0**, return **+0**.
6. Return the result of computing floor(*number*).

16.3.5 10.5.3 Function Declaration Instantiation

NOTE When an execution context is established for evaluating function code a new Declarative Environment Record is created and bindings for each formal parameter, and each function level variable, constant, or function declared in the function are instantiated in the environment record. Formal parameters and functions are initialised as part of this process. All other bindings are initialised during execution of the function code.

Function Declaration Instantiation is performed as follows using arguments *func*, *argumentsList*, and *env*. *func* is the function object that for which the execution context is being established. *env* is the declarative environment record in which bindings are to be created.

1. Let *code* be the value of the [[Code]] internal property of *func*.
2. Let *strict* be the value of the [[Strict]] internal property of *func*.
3. Let *formals* be the value of the [[FormalParameterList]] internal property of *func*.
4. Let *parameterNames* be the BoundNames of *formals*.
5. Let *varDeclarations* be the VarScopedDeclarations of *code*.
6. Let *functionsToInitialise* be an empty List.
7. Let *argumentsObjectNotNeeded* be **false**.
8. For each *d* in *varDeclarations*, in reverse list order do
    a. If *d* is a *FunctionDeclaration* then
        i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
        ii. Let *fn* be the sole element of the BoundNames of *d*.
        iii. If *fn* is "**arguments**", then let *argumentsObjectNotNeeded* be **true**.
        iv. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *fn* as the argument.
        v. If *alreadyDeclared* is **false**, then
            1. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *fn* as the argument.
            2. Assert: *status* is never an Abrupt Completion.
            3. Append *d* to *functionsToInitialise*.
9. For each String *paramName* in *parameterNames*, do
    a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *paramName* as the argument.
    b. NOTE Duplicate parameter names can only occur in non-strict functions. Parameter names that are the same as function declaration names do not get initialised to **undefined**.
    c. If *alreadyDeclared* is **false**, then
        i. If *paramName* is "**arguments**", then let *argumentsObjectNotNeeded* be **true**.
        ii. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *paramName* as the argument.
        iii. Assert: *status* is never an Abrupt Completion
        iv. Call *env*'s InitialiseBinding concrete method passing *paramName*, and **undefined** as the argument.
10. NOTE If there is a function declaration or formal parameter with the name "**arguments**" then an argument object is not created.
11. If *argumentsObjectNotNeeded* is **false**, then

Deleted: ize

Deleted: z

Deleted: z

Deleted: *ize*

Deleted: *ize*

Deleted: z

Deleted: InitializeBinding

Comment [AWB7242]: TODO: don't create an arguments binding for arrow functions (and perhaps for concise methods)

a. If *strict* is **true**, then
    i. Call *env*'s CreateImmutableBinding concrete method passing the String "**arguments**" as the argument.
b. Else,
    i. Call *env*'s CreateMutableBinding concrete method passing the String "**arguments**" as the argument.

12. Let *varNames* be the VarDeclaredNames of *code*.
13. For each String *varName* in *varNames*, in list order do
    a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *varName* as the argument.
    b. NOTE A VarDeclaredNames is only instantiated and initialied here if it is not also the name of a formal parameter or a *FunctionDeclarations*.
    c. If *alreadyDeclared* is **false**, then
        i. Call *env's* CreateMutableBinding concrete method passing *varName* as the argument.
14. Let *lexDeclarations* be the LexicalDeclarations of *code*.
15. For each element *d* in *lexDeclarations* do
    a. NOTE A lexically declared name cannot be the same as a function declaration, formal parameter, or a var name. Lexically declared names are only instantiated here but not initialised.
    b. For each element *dn* of the BoundNames of *d* do
        i. If IsConstantDeclaration of *d* is **true**, then
            1. Call *env*'s CreateImmutableBinding concrete method passing *dn* as the argument.
        ii. Else,
            1. Call *env*'s CreateMutableBinding concrete method passing *dn* and **false** as the arguments.
16. For each *FunctionDeclaration f* in *functionsToInitialise*, do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
    c. Call *env*'s SetMutableMinding concrete method passing *fn*, *fo*, and **false** as the arguments.
17. NOTE Function declaration are initialised prior to parameter initialisation so that default value expressions may reference them. it is not extended code. "**arguments**" is not initialised until after parameter initialization.
18. Let *ao* be the result of InstantiateArgumentsObject with argument *argumentsList*.
19. NOTE If *argumentsObjectNotNeeded* is **true** then the value of *ao* is not directly observable to ECMAScript code and need not actually exist. In that case, its use in the above steps is strictly as a device for specifying formal parameter initialisation semantics.
20. If *argumentsObjectNotNeeded* is **false**, then
    a. If *strict* is **true**, then
        i. Perform the abstract operation CompleteStrictArgumentsObject with argument *a0*.
    b. Else,
        i. Perform the abstract operation CompleteMappedArgumentsObject with arguments *a0*, *func*, *formals*, and *env*.
    c. Call *env*'s InitialiseBinding concrete method passing "**arguments**" and *ao* as arguments.
21. Let *formalStatus* be the result of performing Binding Initialisation for *formals* with *ao* and **undefined** as arguments.
22. ReturnIfAbrupt(*formalStatus*).
23. Return NormalCompletion(empty).

F.1.1 The __proto__ pseudo property.

F.1.1.1 Object.prototype.__proto__

The initial value of the **__proto__** property of the Object prototype object is a data property whose initial value is **null**. This property initially has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

Manipulations of this property as tracked by the Boolean valued primordial internal variable UnderscoreProtoEnabled. The default initial value of UnderscoreProtoEnabled is **true** only if this property is initially present on the primordial Object prototype object.

NOTE Any modification of this property or its attributes causes UnderscoreProtoEnabled to be set to **false**.

Deleted:

Deleted: z

Deleted: *ize*

Deleted: z

Comment [AWB7243]: TODO: don't creat an arguments binding for arrow functions (an perhaps for concise methods)

Deleted: InitializeBinding

Deleted: ..

Comment [AWB244]: The section and algorithm reference in this draft are based up the ES5.1 spec. When the corresponding sections of this document are stable, this section will need to be updated.

Comment [AWB8245]: This materal is goi to be made mandatory and integrated into the main body of the spec.

Comment [AWB246]: This is anticipating r specification material related to Module loade and establishing a primordial environment. basic assumption is that a module loader mus be able to disable this feature. This seems to suggest that the ability to do so must exist in module loader APIs even if this feature is not present in an implementation.

F.1.1.2 Changes To Internal Methods

The definition of the [[Get]] internal method given in 8.12.3 is replaced with the following:

1. If *P* is the string value `"__proto__"` and UnderscoreProtoEnabled is **true**, then
   a. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
   b. If *desc* is not **undefined** and was created by step 1.a to describe the property defined in B.3.1.1 then,
      i. Return the value of the [[Prototype]] internal data property of *O*.
2. Continue by executing the steps of 8.12.3 starting with step 1.

Comment [AWB12247]: This whole thing needs be rewritten, and in the process we will eliminate using [[GetProperty]]

The definition of the [[Put]] internal method given in 8.12.5 is replaced with the following:

Comment [AWB12248]: Need to be updated to use [[SetP]

1. If *P* is the string value `"__proto__"` and UnderscoreProtoEnabled is **true** and *O* is not the standard built-in Object prototype object, then
   a. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
   b. If *desc* is not **undefined** and was created by step 1.a to describe the property defined in B.3.1.1 then,
      i. If the type of *V* is neither Object or Null, return
      ii. Set the value of the [[Prototype]] internal data property of *O* to *V*.
      iii. Return.
2. Continue by executing the steps of 8.12.5 starting with step 1.

Comment [AWB12249]: This whole thing needs to b rewritten, and in the process we will eliminate using [[GetProperty]]

The definition of the [[Delete]] internal method given in 8.12.7 is replaced with the following:

1. If UnderscoreProtoEnabled is **true** and *P* is the string value `"__proto__"` and *O* is the standard built-in Object prototype object, then
   a. Set UnderscoreProtoEnabled to **false**.
2. Continue by executing the steps of 8.12.7 starting with step 1.

The definition of the [[DefineOwnProperty]] internal method given in 8.12.9 is replaced with the following:

Comment [AWB250]: Note that Object.defineOwnPropewrty(obj,'__proto__', desc) nor any other call of [[DefineOwnProperty]] does not modif [[Prototype]].

1. If UnderscoreProtoEnabled is **true** and *P* is the string value `"__proto__"` and *O* is the standard built-in Object prototype object, then
   a. If any attribute contained in *Desc* is not present or has a different value from the corresponding attribute in { [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true** }then,
      i. Set UnderscoreProtoEnabled to **false**.
2. Continue by executing the steps of 8.12.9 starting with step 1.

16.3.5.1 15.18.1.14 Reflect.freeze (target)

When the `freeze` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[Freeze]] internal method of *obj*.

16.3.5.2 15.18.1.15 Reflect.seal (target)

When the `seal` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[Seal]] internal method of *obj*.

16.3.5.3 15.18.1.16 Reflect.isFrozen (target)

When the `isFrozen` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsFrozen]] internal method of *obj*.

16.3.5.4 15.18.1.17 Reflect.isSealed (target)

When the `isSealed` function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsSealed]] internal method of *obj*.

F.1.1.3 __proto__ Object Initialisers

Definitions of two algorithms in 11.1.5 are replaced with the following:

The production *PropertyDefinitionList* **:** *PropertyDefinition* is evaluated as follows:

1. Let *obj* be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
2. Let *propId* be the result of evaluating *PropertyDefinition*.
3. If *propId*.name is the string value `"__proto__"` and UnderscoreProtoEnabled is **true** and IsDataDescriptor(*propId*.descriptor) is **true**, then
   a. Let *v* be *propId*.descriptor.value.
   b. If *desc be propId*.descriptor
   c. If the type of *v* is either Object or Null,
      i. Set the value of the [[Prototype]] internal data property of *obj* to *v*.
   d. Return *obj*.
4. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
5. Return *obj*.

The production
PropertyDefinitionList **:** *PropertyDefinitionList* **,** *PropertyDefinition*
is evaluated as follows:

1. Let *obj* be the result of evaluating *PropertyDefinitionList*.
2. Let *propId* be the result of evaluating *PropertyDefinition*.
3. Let *previous* be the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *propId*.name.
4. If *previous* is not **undefined** then throw a **SyntaxError** exception if any of the following conditions are true
   a. This production is contained in strict code and IsDataDescriptor(*previous*) is **true** and IsDataDescriptor(*propId*.descriptor) is **true**.
   b. IsDataDescriptor(*previous*) is **true** and IsAccessorDescriptor(*propId*.descriptor) is **true.**
   c. IsAccessorDescriptor(*previous*) is **true** and IsDataDescriptor(*propId*.descriptor) is **true**.
   d. IsAccessorDescriptor(*previous*) is **true** and IsAccessorDescriptor(*propId*.descriptor) is **true** and either both *previous* and *propId*.descriptor have [[Get]] fields or both *previous* and *propId*.descriptor have [[Set]] fields
5. If *propId*.name is the string value `"__proto__"` and UnderscoreProtoEnabled is **true** and IsDataDescriptor(*propId*.descriptor) is **true**, then
   a. Let *v* be *propId*.descriptor.value.
   b. If *desc be propId*.descriptor
   c. If the type of *v* is either Object or Null,
      i. Set the value of the [[Prototype]] internal data property of *obj* to *v*.
   d. Return *obj*.

5. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
6. Return *obj*.

Table 40 — Internal Properties Only Defined for Some Objects

Internal Property	*Value Type Domain*	*Description*
[[BuiltinBrand]]	The BuiltinBrand enumeration.	A tag value used by this specification to categorize various kinds of ECMAScript objects defined in this specification.
[[PrimitiveValue]]	*primitive*	Internal state information associated with this object. Of the standard built-in ECMAScript objects, only Boolean, Date, Number, and String objects implement [[PrimitiveValue]].
[[Scope]]	Lexical Environment	A lexical environment that is the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Function objects implement [[Scope]].
[[FormalParameters]]	Parse Tree	A parse tree for ECMAScript code parsed with *FormalParameters* as the goal symbol. Of the standard built-in ECMAScript objects, only Function objects implement [[FormalParameters]].
[[Code]]	Parse Tree	A parse tree for ECMAScript code parsed with *FunctionBody* as the goal symbol. Of the standard built-in ECMAScript objects, only Function objects implement [[Code]].
[[Strict]]	Boolean	**true** if a Function object is a strict mode function. Of the standard built-in ECMAScript objects, only Function objects implement [[Strict]].
[[BoundTargetFunction]]	Object	The target function of a function object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundTargetFunction]] internal property.
[[BoundThis]]	*any*	The pre-bound this value of a function Object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundThis]] internal property.
[[BoundArguments]]	List of *any*	The pre-bound argument values of a function Object created by the standard built-in Function.prototype.bind method. Only objects created by Function.prototype.bind have a [[BoundArguments]] internal property.
[[RegExpMatcher]]	SpecOp(*String*, *index*) → *MatchResult*	Tests for a regular expression match and returns a MatchResult value (see 15.10.2.1). Of the standard built-in ECMAScript objects, only RegExp objects implement [[RegExpMatch]].
[[ParameterMap]]	Object	Provides a mapping between the properties of an arguments object (see 10.6) and the formal parameters of the associated function. Only objects that are arguments objects have a [[ParameterMap]] internal property.

Deleted: 111119
Deleted: NativeBrand
Deleted: Members of t
Deleted: NativeBrand
Deleted: native
Deleted: *FormalParameterList*
Deleted: True if
Deleted: [[Match]]
Deleted: [[Match]]

Bibliography

[1] IEEE Std 754-2008: IEEE Standard for Floating-Point Arithmetic. Institute of Electrical and Electronic Engineers, New York (2008)

[2] The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: The Unicode Standard, Version 3.0 (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5)

[3] ISO 8601:2004(E) *Data elements and interchange formats – Information interchange -- Representation of dates and times*

[4] RFC 1738 "Uniform Resource Locators (URL)", available at <http://tools.ietf.org/html/rfc1738>

[5] RFC 2396 "Uniform Resource Identifiers (URI): Generic Syntax", available at <http://tools.ietf.org/html/rfc2396>

[6] RFC 3629 "UTF-8, a transformation format of ISO 10646", available at <http://tools.ietf.org/html/rfc3629>

[7] RFC 4627 "The application/json Media Type for JavaScript Object Notation (JSON)", available at <http://tools.ietf.org/html/rfc4627>

[8] Unicode Inc. (2010), Unicode Technical Report #15: "Unicode Normalization Forms", available at <http://www.unicode.org/reports/tr15/tr15-29.html>

Deleted: ¶
Unicode Inc. (2010), Unicode Technical Report #15: Unicode Normalization Forms

Deleted: "

Deleted:)",

Deleted: "

Deleted: ",

Deleted: "

Deleted: ",

Deleted: "

Deleted:)"

DRAFT

Formatted Table